

Mariana Miloșescu

C++



Informatică în intensiv

Mariana Miloșescu

XII



MINISTERUL EDUCAȚIEI ȘI CERCETĂRII

Mariana Miloșescu

Informatică intensiv

C++

Filiera teoretică,
profilul real,
specializarea
matematică - informatică,
intensiv informatică

Manual pentru clasa a XI-a



EDITURA DIDACTICĂ ȘI PEDAGOGICĂ, R.A.

Manualul a fost aprobat prin Ordinul ministrului Educației și Cercetării nr. 4742 din 21.07.2006, în urma evaluării calitative organizate de către Consiliul Național pentru Evaluarea și Difuzarea Manualelor și este realizat în conformitate cu programa analitică aprobată prin Ordinul ministrului Educației și Cercetării nr. 3252 din 13.02.2006.

Descrierea CIP a Bibliotecii Naționale a României
MILOȘESCU, MARIANA

Informatică intensiv: manual pentru clasa a XI-a /

Mariana Liliana Miloșescu. - București: Editura Didactică și Pedagogică, 2006
ISBN (10) 973-30-1567-9; ISBN (13) 978-973-30-1567-3

004(075.35)

007(075.35)

Limbajul C ++

© EDP 2006. Toate drepturile asupra acestei ediții sunt rezervate Editurii Didactice și Pedagogice R.A., București. Orice preluare, parțială sau integrală, a textului sau a materialului grafic din această lucrare se face numai cu acordul scris al editurii.

EDITURA DIDACTICĂ ȘI PEDAGOGICĂ, R.A.

Str. Spiru Haret, nr. 12, sector 1, cod 010176, București

Tel.: (021) 315 38 20

Tel./Fax: (021) 312 28 85;
(021) 315 73 98

E-mail: edp@b.astral.ro, edpcom@b.astral.ro

web: www.edituradp.ro

Comenzile pentru această lucrare se primesc:

- prin poștă, pe adresa editurii, cu mențiunea *Comandă carte*
- prin e-mail: edpcom@b.astral.ro, comenzi@edituradp.ro
- prin tel./fax: (021) 315 73 98
- prin tel.: (021) 315 38 20

Referenți: prof. gr. I Emma Gabriela Dornescu,

Colegiul Tehnic Petru Rareș, București; metodist de specialitate – Informatică – I.S.M. București

prof. gr. I Georgina Pătrașcu,

Colegiul Național Sfântul Sava, București

Redactor: Liana Fâcă

Tehnoredactor: Mariana Miloșescu

Desenator: Aurica Georgescu

Coperta: Elena Drăgușel Dumitru

1. Tehnici de programare

1.1. Analiza algoritmilor

Prin **analiza unui algoritm** se identifică resursele necesare pentru executarea algoritmului: **timpul de execuție și memoria**.

Analiza algoritmilor este necesară atunci când există mai mulți algoritmi pentru rezolvarea aceleiași probleme și trebuie ales algoritmul cel mai eficient.

Eficiența unui algoritm este evaluată prin timpul necesar pentru executarea algoritmului.

Pentru a compara – din punct de vedere al eficienței – doi algoritmi care rezolvă aceeași problemă, se folosește aceeași **dimensiune a datelor de intrare – n** (același număr de valori pentru datele de intrare).

Timpul de execuție al algoritmului se exprimă prin numărul de **operații de bază** executate în funcție de **dimensiunea datelor de intrare: T(n)**

Pentru a compara doi algoritmi din punct de vedere al timpului de execuție, trebuie să se stabilească unitatea de măsură care se va folosi, adică **operația de bază** executată în cadrul algoritmilor, după care, se numără de câte ori se execută operația de bază în cazul fiecărui algoritm.

Operația de bază este o operație elementară – sau o succesiune de operații elementare, a căror execuție nu depinde de valorile datelor de intrare.

Există algoritmi la care **timpul de execuție** depinde de **distribuția datelor de intrare**. Să considerăm doi algoritmi de sortare a unui vector cu n elemente – algoritmul de sortare prin metoda selecției directe și algoritmul de sortare prin metoda bulelor – și ca operație de bază **comparăția**. Dacă, în cazul primului algoritm, timpul de execuție nu depinde de distribuția datelor de intrare (modul în care sunt aranjate elementele vectorului înainte de sortarea lui), el fiind $T(n) = \frac{n \times (n - 1)}{2}$, în cazul celui de al doilea algoritm timpul de execuție depinde de distribuția datelor de intrare (numărul de execuții ale structurii repetitive **while** depinde de modul în care sunt aranjate elementele vectorului înainte de sortare). În cazul în care numărul de execuții ale operațiilor elementare depinde de distribuția datelor de intrare, pentru analiza algoritmului se folosesc:

- **timpul maxim de execuție** – timpul de execuție pentru cazul cel mai nefavorabil de distribuție a datelor de intrare; în cazul sortării prin metoda bulelor, cazul cel mai nefavorabil este atunci când elementele vectorului sunt aranjate în ordine inversă decât aceea cerută de criteriul de sortare;
- **timpul mediu de execuție** – media timpilor de execuție pentru fiecare caz de distribuție a datelor de intrare.

Deoarece, în analiza eficienței unui algoritm, se urmărește comportamentul lui pentru o dimensiune mare a datelor de intrare, pentru a compara doi algoritmi din punct de vedere al eficienței, este suficient să se ia în considerare numai factorul care determină **timpul de execuție** – și care este denumit **ordinul de complexitate**.

Ordinul de complexitate al unui algoritm îl reprezintă timpul de execuție – estimat prin ordinul de mărime al numărului de execuții ale operației de bază: $O(f(n))$, unde $f(n)$ reprezintă termenul determinant al timpului de execuție $T(n)$.

De exemplu, dacă – pentru algoritmul de sortare, prin metoda selecției directe – timpul de execuție este $T(n) = \frac{n \times (n - 1)}{2} = \frac{n^2}{2} - \frac{n}{2}$, ordinul de complexitate al algoritmului este $O(n^2)$, deoarece în calcularea lui se ia în considerare numai factorul determinant din timpul de execuție.

În funcție de ordinul de complexitate, există următoarele tipuri de algoritmi:

Ordin de complexitate	Tipul algoritmului
$O(n)$	Algoritm liniar.
$O(n^m)$	Algoritm polinomial. Dacă $m=2$, algoritmul este pătratic, iar dacă $m=3$, algoritmul este cubic.
$O(k^n)$	Algoritm exponentiaș. De exemplu: $2^n, 3^n$ etc. Algoritmul de tip $O(n!)$ este tot de tip exponentiaș, deoarece: $1 \times 2 \times 3 \times 4 \times \dots \times n > 2 \times 2 \times 2 \times \dots \times 2 = 2^{n-1}$.
$O(\log n)$	Algoritm logaritmic.
$O(n \log n)$	Algoritm liniar logaritmic.

De exemplu, algoritmul de sortare prin metoda selecției directe este un algoritm pătratic. Ordinul de complexitate este determinat de structurile repetitive care se execută cu mulțimea de valori pentru datele de intrare. În cazul structurilor repetitive imbricate, ordinul de complexitate este dat de produsul dintre numărul de repetiții ale fiecărei structuri repetitive.

Structura repetitivă	Numărul de execuții ale corpului structurii	Tipul algoritmului
<code>for (i=1; i<=n; i=i+k) { . . . }</code>	$f(n)=n/k \rightarrow O(n)=n$	Liniar
<code>for (i=1; i<=n; i=i*k) { . . . }</code>	$f(n)=\log_k n \rightarrow O(n)=\log n$	Logaritmic
<code>for (i=n; i>=n; i=i/k) { . . . }</code>	$f(n)=\log_k n \rightarrow O(n)=\log n$	Logaritmic
<code>for (i=n; i<=n; i=i+p) { . . . }</code> <code> for (j=n; j<=n; j=j+q) { . . . }</code>	$f(n)=(n/p)*(n/q) = n^2/(p*q) \rightarrow O(n)=n^2$	Polinomial pătratic
<code>for (i=n; i<=n; i=i++) { . . . }</code> <code> for (j=i; j<=n; j=j++) { . . . }</code>	$f(n)=1+2+3+\dots+n=(n*(n+1))/2 \rightarrow O(n)=n^2$	Polinomial pătratic

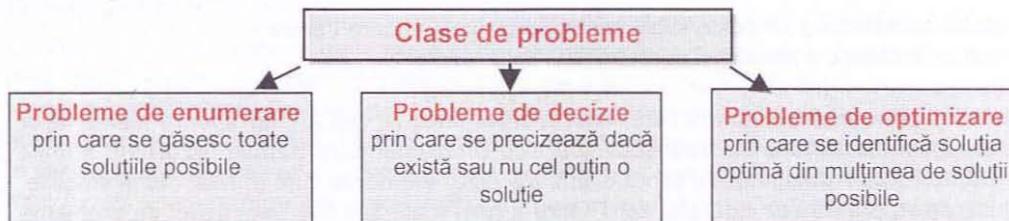
Temă

Determinați complexitatea următorilor algoritmi și precizați tipul algoritmului. Pentru fiecare algoritm se va considera dimensiunea datelor de intrare – n .

- a. determinarea valorii minime dintr-un sir de numere;
- b. inserarea unui element într-un vector, după un element cu valoare precizată;
- c. stergerea dintr-un vector a unui element cu valoare precizată;
- d. stabilirea dacă un sir de numere conține numai numere distincte;
- e. sortarea unui vector folosind metoda bulelor;
- f. căutarea unui element cu valoare precizată, într-un vector nesortat;
- g. căutarea unui element cu valoare precizată, într-un vector sortat;
- h. determinarea tuturor permutărilor unei multimi de numere.

1.2. Metode de construire a algoritmilor

În funcție de procesul de calcul necesar pentru rezolvarea unei probleme, există următoarele clase de probleme:



Generarea tuturor permutărilor unei mulțimi de numere este o problemă de enumerare, căutarea unei valori precizate într-un sir de numere este o problemă de decizie, iar găsirea modalității de plată a unei sume s cu un număr minim de bancnote de valori date este o problemă de optimizare.

Pentru rezolvarea aceleiași probleme se pot folosi mai multe metode de construire a algoritmilor. Ați învățat deja că – pentru rezolvarea aceleiași probleme – puteți folosi un:

- **algoritm iterativ**;
- **algoritm recursiv**.

Soluțiile recursive sunt mult mai clare, mai scurte și mai ușor de urmărit. Alegerea algoritmului recursiv în locul celui iterativ este mai avantajoasă în cazul în care soluțiile problemei sunt definite recursiv sau dacă cerințele problemei sunt formulate recursiv.

Timpul de execuție a unui algoritm recursiv este dat de o formulă recursivă. De exemplu, pentru algoritmul de calculare a sumei primelor n numere naturale, funcția pentru timpul de execuție este prezentată alăturat, unde $\Theta(1)$ reprezintă timpul de execuție a unei operații elementare de atribuire a unei valori sumei. Rezultă că $T(n) = (n+1) \times \Theta(1)$, iar ordinul de complexitate a algoritmului este $O(n)$ la fel ca și cel al algoritmului iterativ. În cazul implementării recursive, fiecare apel al unui subprogram recurrent înseamnă încă o zonă de memorie rezervată pentru execuția subprogramului (variabilele locale și instrucțiunile). Din această cauză, în alegerea între un algoritm iterativ și un algoritm recursiv trebuie ținut cont nu numai de ordinul de complexitate, dar și de faptul că, pentru o adâncime mare a recursivității, algoritmii recursivi nu mai sunt eficienți, deoarece timpul de execuție crește, din cauza timpilor necesari pentru mecanismul de apel și pentru administrarea stivei de sistem.

$$T(n) = \begin{cases} \Theta(1) & \text{pentru } n=0 \\ \Theta(1)+T(n-1) & \text{pentru } n \neq 0 \end{cases}$$

Veți învăța noi **metode de construire** a algoritmilor – care vă oferă avantajul că prezintă fiecare o metodă generală de rezolvare care se poate aplica unei clase de probleme:

- **metoda backtracking**;
- **metoda divide et impera**;
- **metoda greedy**;
- **metoda programării dinamice**.

Fiecare dintre aceste metode de construire a algoritmilor se poate folosi pentru anumite clase de probleme, iar în cazul în care – pentru aceeași clasă de probleme – se pot folosi mai multe metode de construire a algoritmilor, criteriul de alegere va fi **eficiența** algoritmului.

1.3. Metoda backtracking

1.3.1. Descrierea metodei backtracking

Metoda backtracking se poate folosi pentru problemele în care trebuie să se genereze toate soluțiile, o soluție a problemei putând fi dată de un vector:

$$S = \{x_1, x_2, \dots, x_n\}$$

ale cărui elemente aparțin, fiecare, unor multimi finite A_i ($x_i \in A_i$), iar asupra elementelor unei soluții există anumite **restrictii** specifice problemei care trebuie rezolvată, numite **condiții interne**. Multimile A_i sunt multimi ale căror elemente sunt în relații bine stabilită. Multimile A_i pot să coincidă sau nu. Pentru a găsi toate soluțiile unei astfel de probleme folosind o **metodă clasica de rezolvare**, se execută următorul algoritm:

PAS1. Se generează toate elementele produsului cartezian $A_1 \times A_2 \times A_3 \times \dots \times A_n$.

PAS2. Se verifică fiecare element al produsului cartezian, dacă îndeplinește **condițiile interne** impuse ca să fie soluție a problemei.

Studiu de caz

Scop: identificarea problemelor pentru care trebuie enumerate toate soluțiile, fiecare soluție fiind formată din n elemente x_i , care aparțin fiecare unor multimi finite A_i și care trebuie să respecte anumite **condiții interne**.

Enunțul problemei 1: Să se genereze toate permutările multimi $\{1, 2, 3\}$.

Cerința este de a enumera toate posibilitățile de generare a 3 numere naturale din mulțimea $\{1, 2, 3\}$, astfel încât numerele generate să fie distințe (**condiția internă a soluției**). O soluție a acestei probleme va fi un vector cu 3 elemente: $S = \{x_1, x_2, x_3\}$, în care elementul x_i reprezintă numărul care se va găsi, în permutare, pe poziția i , iar mulțimea A_i reprezintă mulțimea numerelor din care se va alege un număr pentru poziția i . În acest exemplu, multimile A_i coincid. Ele au aceleași 3 elemente, fiecare element reprezentând un număr.

$$A_i = \{1, 2, 3\} = A$$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian $A_1 \times A_2 \times A_3 = A \times A \times A = A^3$, adică mulțimea:

$$\{(1,1,1), (1,1,2), (1,1,3), (1,2,1), \dots, (3,3,2), (3,3,3)\}$$

după care se va verifica fiecare element al mulțimii dacă este o soluție a problemei, adică dacă cele trei numere dintr-o soluție sunt distințe. Soluțiile obținute sunt:

$$\{(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1)\}$$

Enunțul problemei 2: Să se genereze toate aranjamentele de 2 elemente ale mulțimi $\{1, 2, 3\}$.

Cerința este de a enumera toate posibilitățile de generare a 2 numere naturale din mulțimea $\{1, 2, 3\}$, astfel încât numerele generate să fie distințe (**condiția internă a soluției**). O soluție a acestei probleme va fi un vector cu 2 elemente: $S = \{x_1, x_2\}$, în care elementul x_i reprezintă numărul care se va găsi în aranjament pe poziția i , iar mulțimea A_i reprezintă mulțimea numerelor din care se va alege un număr pentru poziția i . Își în acest exemplu, multimile A_i coincid. Ele au aceleași 3 elemente, fiecare element reprezentând un număr.

$$A_i = \{1, 2, 3\} = A$$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian $A_1 \times A_2 = A \times A = A^2$, adică mulțimea:

$$\{(1,1), (1, 2), (1,3), (2,1), \dots, (3,2), (3,3)\}$$

după care se va verifica fiecare element al mulțimii, dacă este o soluție a problemei, adică dacă cele două numere dintr-o soluție sunt distincte. Soluțiile obținute sunt:

$$\{(1,2), (1,3), (2,1), (2,3), (3,1), (3,2)\}$$

Enunțul problemei 3: Să se genereze toate combinările de 2 elemente ale mulțimii {1, 2, 3}.

Cerința este de a enumera toate posibilitățile de generare a 2 numere naturale din mulțimea {1,2,3}, astfel încât numerele generate să fie distincte (**condiția internă** a soluției), iar soluțiile obținute să fie distincte. Două soluții sunt considerate distincte dacă nu conțin aceleași numere. O soluție a acestei probleme va fi un vector cu 2 elemente: $S = \{x_1, x_2\}$, în care elementul x_i reprezintă numărul care se va găsi în combinare pe poziția i, iar mulțimea A_i reprezintă mulțimea numerelor din care se va alege un număr pentru poziția i. Își în acest exemplu, mulțimile A_i coincid. Ele au aceleași 3 elemente, fiecare element reprezentând un număr.

$$A_i = \{1, 2, 3\} = A$$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian $A_1 \times A_2 = A \times A = A^2$, adică mulțimea:

$$\{(1,1), (1, 2), (1,3), (2,1), \dots, (3,2), (3,3)\}$$

după care se va verifica fiecare element al mulțimii dacă este o soluție a problemei, adică dacă cele două numere dintr-o soluție sunt distincte și dacă soluția obținută este distinctă de soluțiile obținute anterior. Soluțiile obținute sunt: $\{(1,2), (1,3), (2,3)\}$

Enunțul problemei 4: Să se genereze toate permutările mulțimii {1,2,3,4} care îndeplinesc condiția că 1 nu este vecin cu 3, și 2 nu este vecin cu 4.

Cerința este de a enumera toate posibilitățile de generare a 4 numere naturale din mulțimea {1, 2, 3, 4}, astfel încât numerele generate să fie distincte, iar 1 să nu se învecineze cu 3, și 2 să nu se învecineze cu 4 (**condiția internă** a soluției). O soluție a acestei probleme va fi un vector cu 4 elemente: $S = \{x_1, x_2, x_3, x_4\}$, în care elementul x_i reprezintă numărul care se va găsi în permisie pe poziția i, iar mulțimea A_i reprezintă mulțimea numerelor din care se va alege un număr pentru poziția i. În acest exemplu, mulțimile A_i coincid. Ele au aceleași 4 elemente, fiecare element reprezentând un număr. $A_i = \{1, 2, 3, 4\} = A$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian $A_1 \times A_2 \times A_3 \times A_4 = A \times A \times A \times A = A^4$, adică mulțimea:

$$\{(1,1,1,1), (1,1,1,2), (1,1,1,3), (1,1,1,4), \dots, (4,4,4,3), (4,4,4,4)\}$$

după care se va verifica fiecare element al mulțimii dacă este o soluție a problemei, adică dacă cele patru numere dintr-o soluție sunt distincte și dacă 1 nu se învecinează cu 3, iar 2 cu 4. Soluțiile obținute sunt:

$$\{(1,2,3,4), (1,4,3,2), (2,1,4,3), (2,3,4,1), (3,2,1,4), (3,4,1,2), (4,1,2,3), (4,3,2,1)\}$$

Enunțul problemei 5: Să se aranjeze pe tabla de șah opt dame care nu se atacă între ele (problema celor 8 dame).

Cerința este de a enumera toate posibilitățile de aranjare a 8 dame pe o tablă de șah cu dimensiunea 8x8 (8 linii și 8 coloane), astfel încât toate cele 8 dame să nu se atace între ele (**condiția internă** a soluției). Deoarece nu se pot aranja două dame pe aceeași coloană (s-ar ataca între ele), înseamnă că pe fiecare coloană a tablei de șah se va pune o damă. O soluție a acestei probleme va fi un vector cu 8 elemente, $S = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$, în care elementul x_i reprezintă numărul liniei pe care se va pune dama în coloana i, iar mulțimea A_i reprezintă mulțimea liniilor pe care se poate aranja dama din coloana i. Își în acest caz mulțimile A_i coincid. Ele au aceleași opt elemente, fiecare element reprezentând un număr de linie:

$$A_i = \{1, 2, 3, 4, 5, 6, 7, 8\} = A$$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian $A_1 \times A_2 \times A_3 \times \dots \times A_8 = A \times A \times A \times \dots \times A = A^8$, adică mulțimea:

$\{(1,1,1,1,1,1,1,1), (1,1,1,1,1,1,1,2), (1,1,1,1,1,1,1,3), \dots, (8,8,8,8,8,8,8,7), (8,8,8,8,8,8,8,8)\}$ după care se va verifica fiecare element al mulțimii, dacă este o soluție a problemei, adică dacă cele opt numere dintr-o soluție pot reprezenta coloanele pe care pot fi aranjate damele pe fiecare linie, astfel încât să nu se atace între ele. Soluțiile obținute sunt:

$$\{(1,5,8,6,3,7,2,4), (1,6,8,3,7,4,2,5), (1,7,4,6,8,2,5,3), \dots, (8,3,1,6,2,5,7,4), \\ (8,4,1,3,6,2,7,5)\}$$

Observație Metoda clasică de rezolvare a acestui tip de probleme necesită foarte multe operații din partea calculatorului, pentru a verifica fiecare element al produsului cartezian. Presupunând (pentru simplificare) că fiecare mulțime A_i are m elemente, atunci algoritmul de generare a elementelor produsului cartezian are complexitatea $O(\text{card}(A_1) \times \text{card}(A_2) \times \dots \times \text{card}(A_n)) = O(m \times m \times m \times \dots \times m) = O(m^n)$. Considerând că algoritmul prin care se verifică dacă un element al produsului cartezian este o soluție a problemei (respectă condiția internă a soluției) are complexitatea $O(p)$, atunci **complexitatea algoritmului** de rezolvare a problemei va fi $O(p \times m^n)$. De exemplu, în algoritmul de generare a permutărilor, complexitatea algoritmului de verificare a condiției interne este dată de complexitatea algoritmului prin care se verifică dacă numerele dintr-un sir sunt distincte. În acest algoritm, se parcurge sirul de m numere – și pentru fiecare număr din sir – se parcurge din nou sirul pentru a verifica dacă acel număr mai există în sir. Complexitatea algoritmului este dată de cele două structuri for imbricate: $O(m^2) \rightarrow p = m^2$.



Metoda recomandată pentru acest gen de probleme este metoda **backtracking** sau metoda căutării cu revenire – prin care se reduce volumul operațiilor de găsire a tuturor soluțiilor.

Metoda **backtracking** construiește progresiv vectorul soluției, pornind de la primul element și adăugând la vector următoarele elemente, cu revenire la elementul anterior din vector, în caz de insucces. Elementul care trebuie adăugat se caută în mulțime, printre elementele care respectă condițiile interne.

Prin metoda backtracking se obțin **toate soluțiile problemei**, dacă ele există. Pentru exemplificarea modului în care sunt construite soluțiile, considerăm problema generării permutărilor mulțimii $\{1, 2, 3, \dots, n\}$ ($A_1=A_2= \dots =A_n=A=\{1, 2, 3, \dots, n\}$).

PAS1. Se alege primul element al soluției ca fiind primul element din mulțimea A . În exemplu, $x_1=1$, adică primul număr din permutare este 1.

PAS2. Se caută al doilea element al soluției (x_2). Pentru a-l găsi, se parcurg pe rând elementele mulțimii A și, pentru fiecare element i al mulțimii, se verifică dacă respectă **condițiile interne**. Căutarea continuă până când se găsește primul element din mulțimea A care îndeplinește condiția internă, după care se oprește. În exemplu, se caută numărul de pe a doua poziție a permutării, verificându-se dacă al doilea număr din permutare este diferit de primul număr. Se parcurg primele două elemente ale mulțimii A și se găsește elementul $x_2=2$, după care procesul de căutare se oprește.

PAS3. Se caută al treilea element al soluției (x_3). Căutarea va folosi același algoritm de la Pasul 2. În exemplu, se caută numărul din poziția a treia din permutare. Se găsește elementul $x_3=3$.

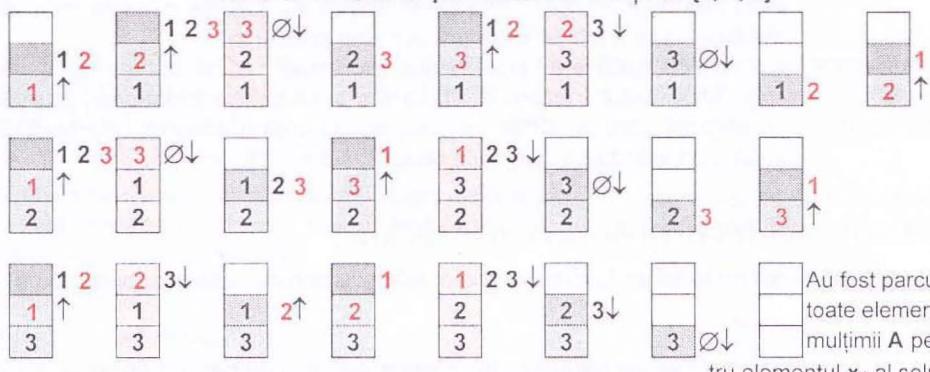
PAS4. Presupunând că s-au găsit primele k elemente ale soluției, $x_1, x_2, x_3, \dots, x_k$, se trece la căutarea celui de al $k+1$ -lea element al soluției, x_{k+1} . Căutarea se va face astfel: se atribuie pe rând lui x_{k+1} elementele mulțimii A , până se găsește primul element i care îndeplinește condiția internă. În exemplu, condiția internă este ca

numărul din poziția $k+1$ a permutării să nu fie egal cu nici unul dintre numerele din pozițiile anterioare lui $k+1$. Pot să apară două situații:

- Există un element i în mulțimea A , astfel încât $x_{k+1} = i$ să fie element al soluției problemei. În acest caz, se atribuie elementului x_{k+1} al soluției valoarea i , după care se verifică dacă s-a găsit soluția problemei. În exemplu, presupunem că pe nivelul $k+1$ s-a găsit numărul 4. Se verifică dacă s-au generat toate cele n elemente ale mulțimii S , adică dacă s-au găsit numere pentru toate cele n poziții din permutare ($k=n$). Dacă s-a găsit soluția problemei, atunci se afișează soluția; altfel, se caută următorul element al soluției, reluându-se operațiile de la Pasul 4.
- S-au parcurs toate elementele mulțimii A și nu s-a găsit nici un element i care să fie elementul x_{k+1} al soluției problemei. Înseamnă că trebuie să revenim la elementul k al soluției – x_k . Așadar, se consideră generate primele $k-1$ elemente ale soluției: x_1, x_2, \dots, x_{k-1} și, pentru elementul x_k al soluției, se reia căutarea, cu următorul element din mulțimea A , adică se reiau operațiile de la Pasul 4 pentru elementul x_k al soluției, însă nu cu primul element din mulțimea A , ci cu elementul din mulțimea A care se găsește imediat după cel care a fost atribuit anterior pentru elementul x_k al soluției. În exemplu, luând în considerare modul în care au fost generate primele k numere ale permutării, în poziția $k+1$, orice număr s-ar alege, el mai există pe una dintre cele k poziții anterioare, și se revine la elementul k , care presupunem că are valoarea 3. Se generează în această poziție următorul număr din mulțimea A (4) și se verifică dacă el nu mai există pe primele $k-1$ poziții ale permutării, iar dacă există, se generează următorul element din mulțimea A (5) și.a.m.d.

PAS 5. Algoritmul se încheie după ce au fost parcuse toate elementele mulțimii A pentru elementul x_1 al soluției. În exemplu, algoritmul se încheie după ce s-au atribuit pe rând valorile 1, 2, ..., n, elementului de pe prima poziție a permutării.

Generarea tuturor permutărilor mulțimii {1, 2, 3}



Observație. În metoda backtracking, dacă s-a găsit elementul x_k al soluției, elementului x_{k+1} al soluției i se atribuie o valoare numai dacă mai există o valoare care să îndeplinească **condiția de continuare** a construirii soluției – adică dacă, prin atribuirea acelei valori, se poate ajunge la o soluție finală pentru care condițiile interne sunt îndeplinite.



Desenați diagramele pentru generarea prin metoda backtracking a:

- tuturor aranjamentelor de 2 elemente ale mulțimii {1, 2, 3};
- tuturor combinărilor de 2 elemente ale mulțimii {1, 2, 3};
- tuturor permutărilor mulțimii {1, 2, 3, 4} care îndeplinesc condiția că 1 nu este vecin cu 3, și 2 nu este vecin cu 4.

Algoritmul metodei **backtracking** poate fi generalizat pentru orice problemă care îndeplinește următoarele condiții:

1. Soluția problemei poate fi pusă sub forma unui vector $S = \{x_1, x_2, \dots, x_n\}$ ale căruia elemente x_i aparțin – fiecare – unei multimi A_i , astfel: $x_1 \in A_1, x_2 \in A_2, \dots, x_n \in A_n$.
2. Multimile A_i sunt finite, iar elementele lor sunt numere întregi și se găsesc într-o ordine bine stabilită.

Algoritmul backtracking este următorul:

PAS1. Se alege primul element al soluției S : $x_1 \in A_1$.

PAS2. Cât timp nu au fost parcuse toate elementele multimi A_1 (nu au fost găsite toate soluțiile) execută:

PAS3. Pentru fiecare element al soluției execută:

PAS4. Se presupune că s-au găsit primele k elemente ale soluției (x_1, x_2, \dots, x_k) aparținând multimilor $A_1, A_2, A_3, \dots, A_k$ și se trece la căutarea celui de al $k+1$ -lea element al soluției, x_{k+1} , printre elementele multimi A_{k+1} . Căutarea se va face astfel: se atribuie, pe rând, lui x_{k+1} , elementele multimi A_{k+1} , până se găsește primul element care îndeplinește **condiția de continuare**.

PAS5. Dacă există un element a_i în multimea A_{k+1} , astfel încât $x_{k+1} = a_i$ să aparțină soluției problemei, atunci se atribuie elementului x_{k+1} valoarea a_i și se trece la Pasul 7; altfel, se trece la Pasul 6.

PAS6. Deoarece s-au parcurs toate elementele multimi A_{k+1} și nu s-a găsit nici un element a_i care să îndeplinească condiția de continuare, se revine la elementul x_k și se consideră generate primele $k-1$ elemente ale soluției: x_1, x_2, \dots, x_{k-1} , și pentru elementul x_k se reia căutarea cu următorul element din multimea A_k , adică se reiau operațiile de la Pasul 4 pentru elementul x_k al soluției, însă nu cu primul element din multimea A_k ci cu elementul din multimea A_k care se găsește imediat după cel care a fost atribuit anterior elementului x_k .

PAS7. Se verifică dacă s-a găsit soluția problemei, adică dacă s-au găsit toate elementele multimi S . Dacă s-a găsit soluția problemei, atunci se afișează soluția; altfel, se trece la căutarea următorului element al soluției, reluându-se operațiile de la Pasul 4.

1.3.2. Implementarea metodei backtracking

Pentru implementarea metodei se folosesc următoarele structuri de date și subprograme.

Date și structuri de date

Pentru a memora elementele x_k ale soluției se folosește o structură de date de tip **stivă**, care este implementată static printr-un vector – st. Pentru vârful stivei se folosește variabila k. Când s-a găsit elementul x_k al soluției, se urcă în stivă, prin incrementarea vârfului cu 1 (k++), pentru a căuta elementul x_{k+1} al soluției, iar pentru a reveni la elementul x_{k-1} al soluției se coboară în stivă prin decrementarea vârfului cu 1 (k--). Inițial, stiva are dimensiunea 1 ($k_{min}=1$), corespunzătoare poziției de pornire, și conține valoarea primului element al soluției. Pentru elementul din vârful stivei (care corespunde elementului x_k al soluției) se va atribui o valoare din multimea A_k care poate fi un element al soluției: st[k]=a[i]. După parcurgerea completă a multimilor A_k , stiva va avea dimensiunea n ($k_{max}=n$) corespunzătoare numărului

de elemente ale soluției. Vârful stivei va fi inițial 1, la găsirea unei soluții va avea valoarea n, iar la terminarea algoritmului vârful stivei va avea valoarea 0.

Se mai folosesc următoarele variabile de memorie:

- **as** – pentru a săi dacă pentru elementul x_k al soluției mai există un succesor, adică dacă mai există un element în mulțimea A_k care ar putea fi elementul x_k al soluției (este o variabilă logică ce are valoarea 1 – true, dacă există succesor; altfel, are valoarea 0 – false),
- **ev** – pentru a săi dacă succesorul găsit respectă **condiția de continuare** și poate fi elementul x_k al soluției (este o variabilă logică ce are valoarea 1 – true, dacă succesorul este element al soluției; altfel, are valoarea 0 – false) și
- **n** – pentru dimensiunea soluției (numărul de elemente ale soluției, în cazul problemelor în care toate soluțiile au același număr de elemente).

```
typedef int stiva[100];
stiva st;           //st=stiva
int n,k,ev,as;    //k=vârful stivei
```

În cazul problemelor prezentate în studiul de caz, un element x_k al soluției este format dintr-o singură valoare: numărul din poziția k (în cazul permutărilor, al aranjamentelor și al combinărilor), respectiv numărul liniei pe care va fi pusă dama din coloana k. În problemele în care trebuie găsit un traseu, un element x_k al soluției este format din două valori care reprezintă coordonatele poziției în care se face următoarea deplasare. În acest caz, pentru memorarea elementelor x_k ale soluției se va folosi o stivă dublă:

```
typedef int stiva[100][3];
stiva st;
```

sau o înregistrare în care cele două câmpuri reprezintă coordonatele deplasării:

```
struct element{int x,y;};
typedef element stiva[100];
stiva st;
```

Elementele mulțimii A_k vor fi perechi de valori (i,j) și vor reprezenta coordonatele unei poziții, iar pentru elementul din vârful stivei se va atribui o valoare, din mulțimea A_k , care poate fi un element al soluției, astfel: $st[k][1]=i$ și $st[k][2]=j$, respectiv $st[k].x=i$ și $st[k].y=j$.

Pentru simplificarea implementării, toate date și structuri de date sunt declarate globale, deoarece valoarea pentru vârful stivei se va transmite mai ușor, între subprograme – ca variabilă globală.

Subprograme

Algoritmul va fi implementat prin:

- un subprogram – care va fi același pentru toți algoritmii de rezolvare prin metoda backtracking (parte fixă) și care descrie **strategia generală backtracking** și
- subprogramele care au **aceeași semnificație** pentru toți algoritmii, dar al căror conținut diferă de la o problemă la alta, depinzând de **condițiile interne** ale soluției.

Semnificația **subprogramelor** folosite este (se va considera ca exemplu generarea permutărilor mulțimii {1, 2, 3, ..., n}):

- Subprogramul **init** (funcție procedurală). Se inițializează elementul din vârful stivei (elementul k). În acest element se va înregistra următorul element al soluției. Acest element se inițializează cu o valoare care nu face parte din mulțimea A_k considerată, urmând ca în următorii pași ai algoritmului să se atrbuie acestui element prima

valoare din mulțimea A_k . În exemplu, nivelul k al stivei se va inițializa cu valoarea 0 ($st[k]=0$), urmând ca la pasul următor să î se atribuie ca valoare 1, adică primul număr din mulțimea {1, 2, 3, ..., n}.

```
void init()
{st[k]=0;}
```

→ Subprogramul **succesor** (funcție operand). Verifică dacă mai există în mulțimea A_k un element pentru nivelul k al soluției (un succesor). Dacă mai există un succesor, se trece la următorul element din mulțimea A_k , iar funcția va returna valoarea 1 (*true*). Dacă nu mai există un succesor, funcția va returna valoarea 0 (*false*). Valoarea returnată de funcție se va atribui variabilei as . Inițial, valoarea variabilei de memorie as este 1 (*true*) – se presupune că mai există un succesor. În exemplu, subprogramul **succesor** va verifica dacă pentru poziția k din permutare mai există un număr. Dacă numărul i de pe nivelul k este mai mic decât n , poziției k î se va atribui numărul următor, $i+1$, și funcția va returna valoarea 1 (*true*), iar dacă numărul de pe nivelul k este n , înseamnă că pe această poziție din permutare nu mai poate fi pus nici un număr – și funcția va returna valoarea 0 (*false*).

```
int successor()
{if (st[k]<n) {st[k]++; return 1;}
else return 0;}
```

→ Subprogramul **valid** (funcție operand). Verifică dacă valoarea atribuită elementului x_k al soluției îndeplinește condiția de continuare, adică poate fi considerată că face parte din soluția problemei (dacă succesorul găsit este element al soluției). Dacă este îndeplinită condiția (se evaluează expresia prin care este descrisă condiția), funcția va returna valoarea 1 (*true*); altfel, va returna valoarea 0 (*false*). Valoarea returnată de funcție se va atribui variabilei ev . Inițial, valoarea variabilei ev este 0 (*false*) – se presupune că succesorul găsit nu este elementul k al soluției. În exemplu, subprogramul **valid** va verifica dacă numărul din poziția k nu mai există în cele $k-1$ poziții anterioare. Dacă numărul nu îndeplinește această condiție, funcția va returna valoarea 0 (*false*).

```
int valid()
{for (int i=1;i<k;i++)
if (st[i]==st[k]) return 0;
return 1;}
```

→ Subprogramul **solutie** (funcție operand). Verifică dacă s-au obținut toate elementele soluției. În exemplu, subprogramul **solutie** va verifica dacă au fost găsite toate cele n elemente ale soluției, adică dacă s-au găsit soluții de aranjare în permutare pentru toate cele n numere. Dacă s-a găsit soluția, subprogramul întoarce valoarea 1 (*true*); altfel, întoarce valoarea 0 (*false*).

```
int solutie()
{return k==n;}
```

→ Subprogramul **tipar** (funcție procedurală). Afisează elementele soluției. De obicei, afișarea soluției constă în afișarea valorilor din stivă.

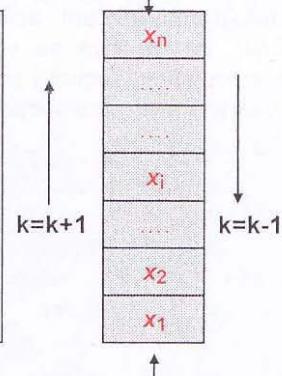
```
void tipar()
{for (int i=1;i<=n;i++) cout<<st[i]<<" ";
cout<<endl;}
```

Numărul de ordine al elementelor soluției

n	a_{n1}	a_{n2}	...	a_{nj}	...	a_{nm}
Parcursarea elementelor se face cu subprogramul succesor						
i	a_{i1}	a_{i2}	...	a_{ij}	...	a_{im}
.....						
2	a_{21}	a_{22}	...	a_{2j}	...	a_{2m}
1	a_{11}	a_{12}	...	a_{1j}	...	a_{1m}
	1	2	...	j	m

Numărul de ordine al elementelor din mulțimea A_k

Elementele soluției



Stiva – $st[k]$

Subprogramul valid verifică dacă este element al soluției

Subprogramul fix poate fi implementat **iterativ** sau **recursiv**.

Implementarea iterativă

```
void bt() //partea fixă a algoritmului
{k=1; //se inițializează vârful stivei
 init(); //se inițializează stiva pentru primul element al soluției
 while (k>0) //cât timp stiva nu s-a golit
 {as=1; ev=0;
  while(as && !ev) //cât timp are succesor și nu s-a găsit
   //elementul k al soluției
   {as=succesor(); //se caută succesor
    if(as) //dacă are succesor, atunci
     ev=valid();} //se verifică dacă este element al soluției
   //se ieșe din structura repetitivă while dacă nu mai există
   //succesor sau dacă s-a găsit elementul soluției
   if(as) //dacă are succesor, atunci
     if (solutie()) //dacă s-au obținut toate elementele soluției,
       tipar(); //atunci se afișează elementele soluției;
     else {k++; //altfel, se urcă în stivă pentru a înregistra
           //următorul element al soluției
           init();} //și se inițializează stiva pentru
                     //următorul element al soluției;
   else k--;} //altfel, se coboară în stivă pentru a reveni
 }
 void main() { ... bt(); ... }
```

Implementarea recursivă – Prelucrările care se fac pentru elementul k al soluției se fac și pentru elementul $k+1$ al soluției și aceste prelucrări pot fi apelate pentru elementul $k+1$ al soluției, iar trecerea de la elementul k al soluției la elementul $k+1$ al soluției se face prin apelul recursiv al acestor prelucrări. În algoritmul backtracking implementat iterativ, revenirea la nivelul $k-1$ trebuie să se facă atunci când pe nivelul k nu se găsește o valoare care să îndeplinească **condițiile interne**. În cazul implementării recursive, **condiția de**

bază este ca pe nivelul k să nu se găsească o valoare care să îndeplinească **condițiile interne**. Când se ajunge la condiția de bază, încețează apelul recursiv și se revine la subprogramul apelant, adică la subprogramul în care se prelucră elementul $k-1$ al soluției, iar în stivă se vor regăsi valorile prelucrate anterior în acest subprogram. Deoarece apelul recursiv se face în funcție de valoarea vârfului stivei (k), această valoare se va transmite, între subprograme, prin intermediul parametrilor de comunicație.

```
void bt(int k) //partea fixă a algoritmului
{init(k); //se inițializează stiva pentru elementul k al soluției
 while(succesor(k))
    //cât timp se găsește succesor pentru elementul k al soluției
    if(valid(k)) dacă succesorul este element al soluției
        if(solutie(k)) //dacă s-au obținut toate elementele soluției,
            tipar(k); //atunci se afișează elementele soluției;
        else bt(k+1); //în sfere, se apelează subprogramul pentru a găsi
    }
    //elementul k+1 al soluției
}
void main() { ... bt(1); ... }
```

Complexitatea algoritmului metodei backtracking

Dacă fiecare soluție are n elemente și fiecare mulțime A_i din care se alege un element al soluției are m elemente, atunci complexitatea algoritmului metodei backtracking este $O(\text{card}(A_1) \times \text{card}(A_2) \times \dots \times \text{card}(A_n)) = O(m \times m \times m \times \dots \times m) = O(m^n)$. Dacă numărul de elemente ale mulțimilor A_i este diferit și notăm cu:

$$\begin{aligned}m_{\min} &= \min(\text{card}(A_1), \text{card}(A_2), \dots, \text{card}(A_n)) \\m_{\max} &= \max(\text{card}(A_1), \text{card}(A_2), \dots, \text{card}(A_n))\end{aligned}$$

atunci complexitatea algoritmului va fi cuprinsă între o complexitate minimă $O(m_{\min}^n)$ și o complexitate maximă $O(m_{\max}^n)$. Rezultă că algoritmul metodei backtracking este un **algoritm exponential**. Având o complexitate exponențială, metoda backtracking se recomandă numai dacă **nu se cunoaște un algoritm mai eficient**.

1.3.3. Probleme rezolvabile prin metoda backtracking

Metoda **backtracking** este recomandată în cazul problemelor care au următoarele caracteristici:

- se cere **găsirea tuturor soluțiilor posibile**;
- nu se cunoaște un algoritm mai eficient.

Alte exemple de probleme clasice care se pot rezolva folosind metoda **backtracking**:

- generarea tuturor elementelor unui produs cartezian;
- generarea tuturor partițiilor unui număr natural;
- generarea tuturor partițiilor unei mulțimi;
- generarea tuturor funcțiilor surjective;
- generarea tuturor funcțiilor injective;
- generarea tuturor posibilităților de plată a unei sume cu bancnote de valori date;
- generarea tuturor posibilităților de acoperire a tablei de șah prin săritura calului (parcurgerea tablei de șah prin săritura calului, fără a se trece de două ori prin aceeași poziție).
- generarea tuturor posibilităților de ieșire dintr-un labirint;

1.3.3.1. Generarea permutărilor

Prin asamblarea subprogramelor definite anterior, programul pentru generarea tuturor permutărilor mulțimii {1, 2, 3, ..., n} va fi:

Implementarea iterativă

```
#include<iostream.h>
typedef int stiva[100];
int n,k,ev,as;
stiva st;
void init()
{st[k]=0;}
int succesor()
{if (st[k]<n)
    {st[k]=st[k]+1; return 1;}
     else return 0;}
int valid()
{for(int i=1;i<k;i++)
  if (st[k]==st[i]) return 0;
return 1;}
int solutie()
{return k==n;}
void tipar()
{for(int i=1;i<=n;i++)
  cout<<st[i]<<" ";
cout<<endl;}
void bt()
{k=1;
init();
while (k>0)
{as=1; ev=0;
while(as && !ev)
  {as=succesor();
   if(as) ev=valid();}
  if(as)
    if (solutie()) tipar();
     else {k++; init();}
  else k--;}
void main()
{cout<<"n= "; cin>>n;
bt();}
```

Implementarea recursivă

```
#include<iostream.h>
typedef int stiva[100];
int n;
stiva st;
void init(int k)
{st[k]=0;}
int succesor(int k)
{if (st[k]<n)
    {st[k]=st[k]+1; return 1;}
     else return 0;}
int valid(int k)
{for(int i=1;i<k;i++)
  if (st[k]==st[i]) return 0;
return 1;}
int solutie(int k)
{return k==n;}
void tipar()
{for(int i=1;i<=n;i++)
  cout<<st[i]<<" ";
cout<<endl;}
void bt(int k)
{init(k);
 while(succesor(k))
   if(valid(k))
     if(solutie(k)) tipar();
      else bt(k+1);}
void main()
{cout<<"n= "; cin>>n;
bt(1);}
```

Algoritmul de generare a permutărilor poate fi folosit și în alte probleme. De exemplu, să se genereze toate permutările mulțimii {1, 2, 3, ..., n} în care nu apar numere consecutive. Această problemă face parte din categoria de probleme de generare a **permutărilor cu condiție** – soluția conține o condiție internă suplimentară față de cea impusă de permutare. În acest exemplu, condiția suplimentară de continuare a construirii soluției este ca numărul ales pentru nivelul **k** al soluției să nu difere printr-o unitate de numărul care se găsește pe nivelul **k-1** al soluției. Modificarea apare în subprogramul **valid()**:

```
int valid()
{if (k>1 && abs(st[k]-st[k-1])==1) return 0;
for (int i=1;i<k;i++) if (st[i]==st[k]) return 0;
return 1;}
```

Temă

Scrieți următoarele programe, în care să folosiți metoda backtracking pentru generarea tuturor permutărilor.

1. Să se genereze toate permutările unei mulțimi de numere oarecare. Numerele se memorează într-un vector. **Indicație.** Se permute indicii elementelor din vectorul **v**, și în subprogramul **tipar()** se afișează elementele **v[st[i]]**.
2. Să se afișeze toate anagramele unui cuvând citit de la tastatură.
3. Să se genereze toate matricele binare pătrate, de dimensiunea **n**, care au un singur element de 1 pe linie și un singur element de 1 pe coloană. O matrice binară este o matrice ale cărei elemente au valoarea 0 sau 1. **Indicație.** Soluția are **n** elemente. Elementul soluției **x_k** reprezintă numărul coloanei de pe linia **k** pe care se găsește elementul cu valoarea 1.
4. Să se genereze toate funcțiile bijective **f : A → B**, unde **card(A)=card(B)=n**.
5. Să se genereze toate posibilitățile de aranjare pe o tablă de șah, cu dimensiunea **n × n**, a **n** ture care să nu se atace între ele. Deoarece tura se poate deplasa numai pe linia sau pe coloana pe care a fost plasată, turele se pot ataca între ele pe linie și pe coloană. **Indicație.** Se observă că fiecare tură trebuie să fie plasată singură pe o coloană, ca să nu se atace între ele. Soluția problemei este dată de mulțimea cu **n** elemente $\{x_1, x_2, \dots, x_n\}$ care se memorează în stivă. Elementul soluției **x_k** reprezintă numărul liniei în care se aşază tura din coloana **k** – și se memorează pe nivelul **k** al stivei **st**. Deoarece două ture nu pot fi așezate pe aceeași linie, stiva trebuie să conțină elemente distincte. Problema se reduce la generarea permutărilor mulțimii $\{1, 2, 3, \dots, n\}$. Interpretarea soluției este: fiecare tură se plasează în coloana **k**, pe linia **st[k]**.
6. Să se genereze toate permutările mulțimii $\{1, 2, 3, \dots, n\}$ în care două numere vecine nu trebuie să fie ambele pare sau ambele impare. **Indicație.** În subprogramul **valid()** se mai verifică și condiția suplimentară de vecinătate.

```
int valid()
{if (k>1 && (st[k-1]%2==0 && st[k]%2==0) ||
     (st[k-1]%2==1 && st[k]%2==1)) return 0;
for (int i=1;i<k;i++) if (st[i]==st[k]) return 0;
return 1;}
```

7. Să se genereze toate permutările mulțimii $\{1, 3, 5, \dots, 2n+1\}$. **Indicație.** Soluția are **n** elemente. În subprogramul **init()** elementul din vârful stivei se initializează cu valoarea -1, iar în subprogramul **succesor()** se modifică modul de determinare a succesorului.

```
int successor()
{if (st[k]<2*n+1) {st[k]=st[k]+2; return 1;}
else return 0;}
```

8. Să se genereze toate permutările unei mulțimi de numere oarecare, astfel încât cea mai mică și cea mai mare valoare să-și păstreze pozițiile inițiale.
9. Într-un șir sunt aranjate **n** persoane. Să se genereze toate posibilitățile de rearanjare în șir a acestor persoane, astfel încât fiecare persoană din șir:
 - a. să nu aibă în față sa aceeași persoană pe care a avut-o în șirul inițial;
 - b. să nu aibă aceeași vecini ca în șirul inițial;

- c. să nu aibă în față sa persoanele pe care le-a avut în sirul inițial;
- d. să fie despărțită – de vecinii pe care i-a avut în sirul inițial – de una sau cel mult p persoane; valoarea pentru p se citește de la tastatură.

1.3.3.2. Generarea produsului cartezian

Se generează toate elementele produsului cartezian $A_1 \times A_2 \times A_3 \times \dots \times A_n$, unde $A_i = \{1, 2, \dots, n_i\}$.

O soluție a problemei va fi un element al produsului cartezian și va fi formată din n elemente. Un element x_k al soluției poate fi orice element din mulțimea A_k (nu există condiții interne). Numărul de elemente ale fiecărei mulțimi A_i va fi memorat într-un vector m cu lungimea logică n, unde $m[i] = n_i$. Față de algoritmul pentru generarea permutărilor, apar următoarele diferențe:

- Deoarece fiecare mulțime A_k are un număr diferit de elemente, elementul de pe nivelul k are succesor dacă numărul i de pe acest nivel este mai mic decât $m[k]$.
- Deoarece nu există condiții interne, nu există restricții nici pentru condiția de continuare, și subprogramul valid() va furniza valoarea 1.

Implementarea iterativă

```
#include<iostream.h>
typedef int stiva[10];
int n,k,ev,as,m[10];
stiva st;
void init()
{st[k]=0;}
int succesor()
{if (st[k]<m[k])
    {st[k]=st[k]+1; return 1;}
     else return 0;}
int valid()
{return 1;}
int solutie()
{return k==n;}
void tipar()
{for(int i=1;i<=n;i++)
    cout<<st[i]<<" ";
cout<<endl;}
void bt()
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n;
 for(int i=1;i<=n;i++)
 {cout<<"Nr. de elemente multimea "
    <<i<<" ";
    cin>>m[i];}
bt();}
```

Implementarea recursivă

```
#include<iostream.h>
typedef int stiva[10];
int n;
stiva st;
void init(int k)
{st[k]=0;}
int succesor(int k)
{if (st[k]<m[k])
    {st[k]=st[k]+1; return 1;}
     else return 0;}
int valid(int k)
{return 1;}
int solutie(int k)
{return k==n;}
void tipar()
{for(int i=1;i<=n;i++)
    cout<<st[i]<<" ";
cout<<endl;}
void bt(int k)
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n;
 for(int i=1;i<=n;i++)
 {cout<<"Nr. de elemente multimea "
    <<i<<" ";
    cin>>m[i];}
bt(1);}
```

Algoritmul de generare a produsului cartezian poate fi folosit și în alte probleme. De exemplu, pentru **generarea tuturor submulțimilor unei mulțimi**.

O submulțime este formată din elemente ale mulțimii A. Pentru simplificarea algoritmului, vom considera mulțimea A={1,2,3,...,n}. Considerăm mulțimea B={0,1}. Pentru construirea submulțimilor, pentru fiecare submulțime se definește funcția $f:A \rightarrow B$, astfel: dacă elementul i din mulțimea A aparține submulțimii, atunci $f(i)=1$; altfel, $f(i)=0$. Problema se reduce la generarea produsului cartezian B^n . Soluția va fi memorată în stivă și va avea n elemente, fiecare element k al soluției având valoarea 0 sau 1, cu semnificația că elementul k din mulțimea A aparține, respectiv nu aparține submulțimii. Alăturat, sunt prezentate toate submulțimile mulțimii {1,2,3} și conținutul stivei pentru fiecare dintre ele.

Submulțimile	Stiva
$\emptyset = \emptyset$	0 0 0
{3}	0 0 1
{2}	0 1 0
{2, 3}	0 1 1
{1}	1 0 0
{1, 3}	1 0 1
{1, 2}	1 1 0
{1, 2, 3}	1 1 1

Față de programul anterior, apar următoarele modificări:

- Deoarece mulțimile produsului cartezian sunt identice ($B=\{0,1\}$), s-au modificat: subprogramul init() – inițializarea nivelului k al stivei se face cu valoarea -1, cu 1 mai mic decât 0 – și subprogramul succesor() – elementul are succesor numai dacă este mai mic decât ultimul element din mulțime, 1.
- În subprogramul de afișare a soluției, va fi afișat numărul nivelului i pentru care, în stivă, se memorează valoarea 1.

Implementarea iterativă

```
#include<iostream.h>
typedef int stiva[100];
int n,k,ev,as;
stiva st;
void init()
{st[k]=-1;}
int succesor()
{if (st[k]<1)
    {st[k]=st[k]+1; return 1;}
     else return 0;}
int valid()
{return 1;}
int solutie()
{return k==n;}
void tipar ()
{int i,x=0; cout<<"(";
 for (i=1;i<=n;i++)
 if (st[i]==1)
 {cout<<i<<", "; x=1;}
 if (x) cout<<'\'b';
 cout<<") "<<endl;}
void bt()
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n;
bt();}
```

Implementarea recursivă

```
#include<iostream.h>
typedef int stiva[100];
int n;
stiva st;
void init (int k)
{st[k]=0;}
int succesor(int k)
{if (st[k]<st[k-1]+1)
    {st[k]=st[k]+1;return 1;}
     else return 0;}
int valid() {return 1;}
int solutie(int k)
{return k==n;}
void tipar()
{int i,x=0; cout<<"(";
 for (i=1;i<=n;i++)
 if (st[i]==1)
 {cout<<i<<", "; x=1;}
 if (x) cout<<'\'b';
 cout<<") "<<endl;}
cout<<endl;}
void bt(int k)
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n; bt(1);}
Observație. Caracterul escape '\b' este caracterul Backspace (sterge ultimul caracter din sir)
```

Temă

Scriți următoarele programe, în care să folosiți metoda backtracking pentru generarea produsului cartezian.

1. Se consideră n multimi de cuvinte A_i , fiecare multime având n_i cuvinte. Să se genereze toate textele de n cuvinte care se pot forma cu cuvintele din aceste multimi, cuvântul i din text aparținând multimii A_i .
2. Într-un restaurant, un meniu este format din trei feluri de mâncare. Există patru preparate culinare pentru felul unu, cinci preparate culinare pentru felul doi și trei preparate culinare pentru felul 3. Să se genereze toate meniurile care se pot forma cu aceste preparate culinare.
3. Într-un restaurant, un meniu este format din trei feluri de mâncare. Există patru preparate culinare pentru felul unu, cinci preparate culinare pentru felul doi și trei preparate culinare pentru felul 3. Fiecare preparat culinar are un preț și un număr de calorii. Să se genereze toate meniurile pe care le poate comanda o persoană, care să nu depășească suma s și numărul de calorii c . Datele se citesc dintr-un fișier text, astfel: de pe primul rând suma s și numărul de calorii, de pe rândul următor, în ordine, prețul fiecărui preparat culinar, și de pe ultimul rând, în aceeași ordine, calorile fiecărui preparat culinar.
4. Pentru o multime oarecare A , cu n elemente, să se genereze toate submultimile care au suma elementelor egală cu s . Numărul de elemente ale multimii, elementele multimii A și valoarea pentru suma s se citesc de la tastatură.
5. Să se afișeze toate numerele cu n cifre ($1 \leq n \leq 10$) care au proprietatea că sunt formate numai din cifre pare, în ordine descrescătoare.
6. Să se afișeze toate numerele formate din cifre distincte cu proprietatea că suma cifrelor este S . Valoarea variabilei S se citește de la tastatură.
7. Să se afișeze toate secvențele de n litere (n număr natural par, citit de la tastatură) din multimea $\{A, B, C, D\}$, secvențe care se construiesc astfel: nu se aşază două litere identice una lângă alta și trebuie să se folosească exact $n/2$ litere A.
8. Se citesc de la tastatură un număr natural n ($0 < n \leq 10$) și apoi n numere naturale $a_1, a_2, a_3, \dots, a_n$. Să se afișeze pe ecran toate posibilitățile de a intercală între toate cele n numere operatorii + și - astfel încât, evaluând expresia obținută, de la stânga la dreapta, la fiecare pas, rezultatul obținut să fie strict pozitiv.
9. Să se rezolve – în multimea numerelor naturale – ecuația $4x+3y+2xy=48$. **Indicație.** Soluția are 2 elemente: x și y . Ele pot lua valori în intervalul $[0, 16]$. Limita inferioară a intervalului este 0, pentru că numerele sunt naturale. Limita superioară s-a determinat considerând, pe rând, situațiile limită $x=0$ și $y=0$. Considerăm multimea $A=\{0, 2, 3, \dots, 16\}$. Problema se reduce la generarea produsului cartezian cu condiție $A \times A$ – soluția conține o condiție internă suplimentară: elementele ei trebuie să verifice ecuația.
10. Se citesc n cifre distincte și un număr natural x . Să se genereze toate numerele care se pot forma cu aceste cifre și sunt mai mici decât numărul x . De exemplu, pentru cifrele 0, 1 și 3 și numărul $x=157$, se generează 1, 3, 10, 11, 13, 30, 31, 33, 100, 101, 103, 110, 111, 113, 130, 131, 133. **Indicație.** Se calculează numărul de cifre m ale numărului x . Pentru multimea A formată din cele n cifre, se generează produsele carteziene A^P , cu $1 \leq p \leq m$. Elementele produsului cartezian sunt cifrele numărului care se formează. Pentru ca un element al produsului cartezian să fie soluție, trebuie ca primul element să fie diferit de 0 (cifra cea mai semnificativă din număr nu trebuie să fie 0), iar numărul format cu m cifre să fie mai mic decât numărul x .
11. Să se genereze toate numerele naturale, cu cel mult n cifre ($n \leq 10$), care sunt formate numai din cifre pare, în ordine strict crescătoare.

12. Să se genereze toate numerele naturale, cu n cifre, care conțin p cifre k . Valorile pentru n , p și k se citesc de la tastatură.
13. Se citește un număr natural n . Să se genereze toate numerele naturale care, reprezentate în baza 2, au același număr de cifre de 0 și același număr de cifre de 1 ca și reprezentarea în baza 2 a numărului n .
14. Pe un bilet există 12 poziții care pot fi perforate, aranjate pe 4 linii și 3 coloane. Să se genereze toate posibilitățile de perforare a celor 12 poziții, astfel încât să nu existe două poziții alăturate neperforate. **Indicație.** Considerăm mulțimea A , formată din 12 elemente, fiecare element reprezentând o poziție care poate fi perforată, și mulțimea $B=\{0,1\}$. Se definește funcția $f:A \rightarrow B$ astfel: dacă poziția i este perforată, atunci $f(i)=1$; altfel, $f(i)=0$. Problema se reduce la generarea produsului cartezian B^{12} – soluția conține o condiție internă suplimentară: poziția k care se adaugă, nu trebuie să fie neperforată (nu trebuie să aibă valoarea 0) dacă poziția $k-1$ sau poziția $k-3$ este neperforată (are valoarea 0).

1.3.3.3. Generarea aranjamentelor

Se generează toate aranjamentele de m elemente ale mulțimii $\{1, 2, 3, \dots, n\}$.

O soluție a problemei va fi un aranjament – și va avea m elemente. Față de algoritmul pentru generarea permutărilor, se modifică doar condiția prin care se verifică dacă s-au obținut toate elementele soluției.

Implementarea iterativă

```
#include<iostream.h>
typedef int stiva[100];
int n,m,k,ev,as;
stiva st;
void init()
{st[k]=0;}
int succesor()
{if (st[k]<n)
    {st[k]=st[k]+1; return 1;}
     else return 0;}
int valid()
{for(int i=1;i<k;i++)
    if (st[k]==st[i]) return 0;
return 1;}
int solutie()
{return k==m;}
void tipar()
{for(int i=1;i<=m;i++)
    cout<<st[i]<<" ";
cout<<endl;}
void bt()
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n;
 cout<<"m= "; cin>>m; bt();}
```

Implementarea recursivă

```
#include<iostream.h>
typedef int stiva[100];
int n,m;
stiva st;
void init(int k)
{st[k]=0;}
int succesor(int k)
{if (st[k]<n)
    {st[k]=st[k]+1; return 1;}
     else return 0;}
int valid(int k)
{for(int i=1;i<k;i++)
    if (st[k]==st[i]) return 0;
return 1;}
int solutie(int k)
{return k==m;}
void tipar()
{for(int i=1;i<=m;i++)
    cout<<st[i]<<" ";
cout<<endl;}
void bt(int k)
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n;
 cout<<"m= "; cin>>m; bt(1);}
```

Algoritmul de generare a aranjamentelor poate fi folosit și în alte probleme. De exemplu, pentru **generarea tuturor funcțiilor injective**.

Se generează toate funcțiile injective $f:A \rightarrow B$, unde $\text{card}(A)=m$ și $\text{card}(B)=n$. Pentru simplificarea algoritmului vom considera mulțimile $A=\{1,2,3,\dots,m\}$ și $B=\{1,2,3,\dots,n\}$. O soluție este formată din m elemente. Elementul k al soluției reprezintă valoarea funcției $f(k)$. Deoarece valoarea funcției $f(k)$ aparține mulțimii B , în stivă se vor genera numere din mulțimea $\{1,2,3,\dots,n\}$. Din definiția funcției injective, $f(i) \neq f(j)$, pentru orice $i \neq j$. Rezultă că, pentru ca funcția să fie injectivă, trebuie ca $m \leq n$. Problema se reduce la generarea în stivă a tuturor aranjamentelor de n elemente luate câte m . Soluțiile vor fi afișate sub forma tabelului de variație al funcției. De exemplu, dacă $A=\{1,2\}$ și $B=\{1,2,3\}$, soluțiile și modul în care vor fi afișate sunt prezentate alăturat.

Față de programul anterior, nu se va modifica decât subprogramul de afișare a soluțiilor **tipar()**.

```
void tipar()
{
    int i; cout << " x | ";
    for (i=1;i<=m;i++) cout << i << " "; cout << endl;
    for (i=1;i<=m;i++) cout << "-----"; cout << endl << "f(x) | ";
    for (i=1;i<=m;i++) cout << st[i] << " "; cout << endl << endl;
}

void main()
{cout << "numărul de elemente ale multimii A= "; cin >> m;
 cout << "numărul de elemente ale multimii B= "; cin >> n; bt();}
```


Tema

Scrieți următoarele programe, în care să folosiți metoda backtracking pentru generarea tuturor aranjamentelor.

1. Să se genereze toate posibilitățile de aranjare pe m scaune a n persoane ($m \leq n$). Valorile pentru n și m și numele persoanelor se citesc dintr-un fișier text.
2. Se citesc de la tastatură n cifre distințe. Să se genereze toate numerele de m cifre ($m \leq n$) care se pot forma cu aceste cifre – și care conțin toate cele n cifre.
3. Să se genereze toate anagramele unui cuvânt, din care se elimină p litere oarecare. Valoarea minimă a numărului p este 1, iar valoarea maximă este cu 1 mai mică decât lungimea cuvântului. Se citesc de la tastatură cuvântul și valoarea numărului p .
4. Se citesc de la tastatură n caractere distințe. Să se genereze toate cuvintele de m caractere ($m \leq n$) care se pot forma cu aceste caractere și care conțin toate cele n caractere.
5. Se citește un număr n care are m cifre. Să se genereze toate numerele, cu cel mult m cifre, care se pot forma cu cifrele numărului inițial.
6. Dintr-o mulțime de n persoane, aranjate într-un sir, se elimină p persoane. Să se genereze toate posibilitățile de aranjare într-un sir a persoanelor rămase, astfel încât fiecare persoană să nu aibă aceeași vecină ca în sirul inițial. Valorile pentru n și p se citesc de la tastatură.
7. Să se genereze toate aranjamentele, de m numere, ale unei mulțimi de n numere oarecare astfel încât suma elementelor generate să fie egală cu s . Numărul de elemente ale mulțimii, n , elementele mulțimii, valoarile pentru m și pentru suma s , se citesc de la tastatură.
8. Să se genereze toate drapelele cu trei culori care se pot forma cu șase culori: alb, galben, roșu, verde, albastru și negru, care au în mijloc culoarea alb, verde sau roșu. **Indicație.** Soluția are 3 elemente, un element al soluției fiind indicele din vector al unei

Soluția	Afișarea
1 2	x 1 2
	f(x) 1 2
1 3	x 1 2
	f(x) 1 3
2 1	x 1 2
	f(x) 2 1
2 3	x 1 2
	f(x) 2 3
3 1	x 1 2
	f(x) 3 1
3 2	x 1 2
	f(x) 3 2

culori. Se generează aranjamente cu condiție de 6 obiecte luate câte 3 – soluția conține o condiție internă suplimentară față de cea impusă de aranjamente: elementul 2 al soluției trebuie să fie indicele culorii alb, verde sau roșu.

9. Se consideră n cuburi. Fiecare cub i are latura L_i și culoarea c_i . Să se construiască toate turnurile stable, formate cu m cuburi, astfel încât două cuburi vecine în turn să nu aibă aceeași culoare. Valorile pentru n și m și atributele celor n cuburi se citesc de la tastatură. **Indicație.** Informațiile despre cuburi se memorează într-un vector de înregistrări cu n elemente. Soluția are m elemente. Un element al soluției este indicele din vector al unui cub. Se generează aranjamente cu condiție de n obiecte luate câte m – soluția conține o condiție internă suplimentară față de cea impusă de aranjamente: cubul k care se adaugă la turn trebuie să aibă latura mai mică decât a cubului $k-1$ și culoarea diferită de acesta.

1.3.3.4. Generarea combinațiilor

Se generează toate combinațiile de m elemente ale mulțimii $\{1, 2, 3, \dots, n\}$.

O soluție a problemei va fi o combinare și va avea m elemente. Față de algoritmul pentru generarea aranjamentelor, apare o condiție suplimentară: aceea că soluțiile obținute să fie distincte, adică două soluții să nu conțină aceleași numere. Pentru aceasta, se va adăuga o condiție de continuare suplimentară: valoarea de pe nivelul k trebuie să fie strict mai mare decât oricare dintre valorile de pe nivelele inferioare. Altfel spus, elementele soluției trebuie să fie ordonate: $st[1] < st[2] < \dots < st[k-1] < st[k]$. Condiția de continuare este îndeplinită dacă elementul de pe nivelul k va avea o valoare strict mai mare decât valoarea elementului de pe nivelul $k-1$ (se inițializează cu o valoare egală cu cea a elementului de pe nivelul $k-1$) și o valoare mai mică decât $n-m+k$ (se caută succesorul până la valoarea $n-m+k$).

Implementarea iterativă

```
#include<iostream.h>
typedef int stiva[100];
int n,m,k,ev,as;
stiva st;
void init()
{if (k==1) st[k]=0;
 else st[k]=st[k-1];}
int successor()
{if (st[k]<n-m+k)
 {st[k]=st[k]+1; return 1;}
 else return 0;}
int valid() {return 1;}
int solutie()
{return k==m;}
void tipar()
{for(int i=1;i<=m;i++)
 cout<<st[i]<<" ";
 cout<<endl;}
void bt()
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n;
 cout<<"m= "; cin>>m; bt();}
```

Implementarea recursivă

```
#include<iostream.h>
typedef int stiva[100];
int n,m;
stiva st;
void init(int k)
{ if (k==1) st[k]=0;
 else st[k]=st[k-1];}
int successor(int k)
{if (st[k]<n-m+k)
 {st[k]=st[k]+1; return 1;}
 else return 0;}
int valid() {return 1;}
int solutie(int k)
{return k==m;}
void tipar()
{for(int i=1;i<=m;i++)
 cout<<st[i]<<" ";
 cout<<endl;}
void bt(int k)
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n;
 cout<<"m= "; cin>>m; bt(1);}
```

Algoritmul de generare a combinațiilor poate fi folosit în problemele de **generare a tuturor posibilităților de a forma din n obiecte grupuri de m obiecte care să aibă anumite proprietăți**. De exemplu, din n obiecte trebuie să se distribuie unei persoane m obiecte, care să conțină obligatoriu obiectul p și să nu conțină obiectul q . Să se genereze toate posibilitățile de a forma grupuri de m astfel de obiecte. Pentru simplificare vom considera multimea obiectelor ca fiind $\{1, 2, 3, \dots, n\}$. Valorile pentru numărul de obiecte n , numărul de obiecte din grup, m , și indicii obiectelor p și q – se citesc de la tastatură.

Deoarece obiectul p face parte obligatoriu din grupul de obiecte, el va fi memorat ca prim element al soluției ($st[1]=p$), iar subprogramul $bt()$ va genera numai următoarele $m-1$ elemente ale soluției. Se modifică și condiția de continuare a construirii soluției (subprogramul $valid()$), deoarece obiectele p și q nu pot fi elemente ale soluției.

```
#include<iostream.h>
typedef int stiva[100];
int n,m,p,q,k,ev,as;
stiva st;
void init() {if (k==2) st[k]=0; else st[k]=st[k-1];}
int succesor() {//la fel ca în exemplul anterior}
int valid()
{return st[k]!=p && st[k]!=q;}
int solutie() {//la fel ca în exemplul anterior}
void tipar() {//la fel ca în exemplul anterior}
void bt()
{k=2;init();
 while (k>1)
//la fel ca în exemplul anterior}
void main()
{cout<<"n= "; cin>>n; cout<<"m= "; cin>>m;
 cout<<"p= "; cin>>p; cout<<"q= "; cin>>q;
 st[1]=p; bt();}
```



Scrieți următoarele programe, în care să folosiți metoda backtracking pentru generarea tuturor combinațiilor.

1. Să se genereze toate grupurile de p persoane care se pot forma din n persoane. Informațiile se citesc dintr-un fișier text unde, pe primul rând, sunt scrise valorile pentru p și n , despărțite printr-un spațiu, iar pe următoarele rânduri numele persoanelor, câte un nume pe un rând.
2. Două persoane își împart n obiecte astfel: prima persoană ia m obiecte, iar cealaltă persoană – restul obiectelor. Să se genereze toate posibilitățile de distribuire a celor n obiecte între cele două persoane.
3. Două persoane își împart n obiecte. Fiecare obiect are o valoare v_i . Să se genereze toate posibilitățile de distribuire a celor n obiecte, între cele două persoane, astfel încât fiecare persoană să primească obiecte, care să aibă aceeași valoare totală.
4. Două persoane își împart n obiecte astfel: prima persoană ia m obiecte, iar cealaltă persoană, restul obiectelor. Fiecare obiect are o valoare v_i . Să se genereze toate posibilitățile de distribuire a celor n obiecte, între cele două persoane, astfel încât fiecare persoană să primească obiecte care să aibă aceeași valoare totală.

5. Să se genereze toate numerele binare de n cifre care au m cifre de 0. Indicație. O soluție este formată din m elemente, un element fiind poziția din număr în care se poate găsi cifra 0 – exceptând prima poziție.
6. Din n obiecte date, să se genereze toate grupurile de m obiecte care îndeplinesc următoarele condiții:
- conțin exact p obiecte precizate;
 - nu conțin nici unul din p obiecte precizate;
 - conțin numai un obiect din p obiecte precizate;
 - conțin cel puțin un obiect din p obiecte precizate;
 - conțin exact q obiecte din p obiecte precizate;
 - conțin exact q obiecte din p obiecte precizate și nu conțin r obiecte precizate;
 - conțin exact q obiecte din p obiecte precizate și nu conțin s obiecte din r obiecte precizate.

1.3.3.5. Generarea tuturor partițiilor unui număr natural

O partitie a unui număr natural nenul n este o descompunere a numărului n în sumă de m numere naturale nenule. Soluțiile care nu diferă decât prin ordinea termenilor nu sunt considerate distințe. Alăturat sunt prezentate toate partițiile numărului 4.

$4=1+1+1+1$ ($m=4$)
$4=1+1+2$ ($m=3$)
$4=1+3$ ($m=2$)
$4=2+2$ ($m=2$)
$4=4$ ($m=1$)

O soluție a problemei va fi formată din m elemente a căror sumă este egală cu numărul n . Elementele soluției reprezintă termenii unei descompuneri. Numărul de elemente ale unei soluții (m) este egal cu valoarea vârfului stivei k atunci când s-a obținut soluția. Soluția se obține atunci când suma elementelor din stivă este egală cu n . Altfel spus, soluția se obține atunci când suma $s=st[1]+st[2]+\dots+st[k-1]+st[k]$ are valoarea n . Pentru a evita repetarea aceleiași descompunerii, valoarea de pe nivelul k trebuie să fie mai mare sau egală cu oricare dintre valorile de pe nivelele inferioare. Altfel spus, elementele soluției trebuie să fie ordonate: $st[1]\leq st[2]\leq\dots\leq st[k-1]\leq st[k]$. Această condiție este îndeplinită dacă elementul de pe nivelul k va avea o valoare mai mare sau egală cu valoarea elementului de pe nivelul $k-1$ (se inițializează cu o valoare mai mică cu 1 decât cea a elementului de pe nivelul $k-1$) și o valoare mai mică decât diferența dintre numărul n și suma termenilor descompunerii găsiți până la nivelul k , $s=st[1]+st[2]+\dots+st[k-1]$.

Condiția de continuare este ca suma termenilor generați, s , să fie mai mică sau egală cu n ($s\leq n$), o soluție completă obținându-se atunci când $s=n$. Inițial, suma s are valoarea 0 și ea trebuie actualizată în permanență. Există două cazuri de actualizare:

→ Atunci când se adaugă în stivă un nou termen, acesta se adaugă și la sumă. Altfel spus, dacă succesorul găsit poate fi element al soluției (prin adăugarea lui la sumă aceasta nu va depăși valoarea numărului n), atunci el se adaugă la sumă. Actualizarea sumei se va face în subprogramul `valid()` (termenul se adaugă la sumă numai dacă face parte din descompunere): $s+=st[k]$.

→ Atunci când se coboară în stivă, trebuie scăzute din sumă două valori: valoarea elementului din vârful stivei (k) și valoarea elementului de pe nivelul precedent ($k-1$). Deoarece în stivă se coboară întotdeauna după ce s-a obținut o soluție completă și nu se mai poate găsi un element pentru nivelul k cu care să se continue dezvoltarea soluției, scăderea din sumă a termenului curent se va face după ce se afișează o soluție completă în subprogramul `tipar()`: $s-=st[k]$, iar scăderea din sumă a

termenului precedent se face în subprogramul `succesor()`, atunci când nu se găsește succesor pentru elementul din vârful stivei: $- s == st[k-1]$.

Implementarea iterativă

```
#include<iostream.h>
typedef int stiva[100];
int n,s=0,k,ev,as;
stiva st;
void init ()
{if (k==1) st[k]=0;
 else st[k]=st[k-1]-1;}
int succesor()
{if (st[k]<n-s)
 {st[k]=st[k]+1; return 1;}
 else {s-=st[k-1]; return 0;}}
int valid()
{if (s+st[k]<=n)
 {s+=st[k]; return 1;}
 else return 0;}
int solutie()
{return s==n;}
void tipar()
{for(int i=1;<k;i++)
 cout<<st[i]<<" ";
 cout<<st[k]<<endl;
 s-=st[k];}
void bt()
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n; bt();}
```

Implementarea recursivă

```
#include<iostream.h>
typedef int stiva[100];
int n,s=0;
stiva st;
void init (int k)
{if (k==1) st[k]=0;
 else st[k]=st[k-1]-1;}
int succesor(int k)
{if (st[k]<n-s)
 {st[k]=st[k]+1; return 1;}
 else {s-=st[k-1]; return 0;}}
int valid(int k)
{if (s+st[k]<=n)
 {s+=st[k]; return 1;}
 else return 0;}
int solutie(int k)
{return s==n;}
void tipar(int k)
{for(int i=1;<k;i++)
 cout<<st[i]<<" ";
 cout<<st[k]<<endl;
 s-=st[k];}
void bt(int k)
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n; bt(1);}
```

Algoritmul de generare a tuturor partițiilor unui număr natural poate fi folosit și în alte probleme. De exemplu, **generarea tuturor posibilităților de plată a unei sume cu bancnote de valori date**.

Problema se reduce la generarea tuturor partițiilor unui număr natural nenul **suma**, o parte fiind o descompunere a numărului **suma** în sumă de **m** numere naturale, nenule, cu valori aparținând mulțimii $A=\{a_1, a_2, \dots, a_n\}$. Alăturat, sunt prezentate toate partițiile sumei 25 pentru bancnote cu valori aparținând mulțimii $A=\{5, 10\}$.

$$\begin{aligned} 25 &= 5+5+5+5+5 \quad (m=5) \\ 25 &= 5+5+5+10 \quad (m=4) \\ 25 &= 5+10+10 \quad (m=3) \end{aligned}$$

Față de problema anterioară, apar următoarele modificări:

- Deoarece suma **suma** se descompune în valori precizate, aparținând mulțimii **A**, aceste valori se memorează în vectorul **a**, care are lungimea logică **n** (numărul de valori ale bancnotelor).
- Un element al soluției este indicele valorii bancnotei din vectorul **a**, adică un element din mulțimea $\{1, 2, \dots, n\}$.
- O soluție a problemei va fi formată din **m** elemente, alese astfel încât suma valorilor bancnotelor corespunzătoare lor să fie egală cu numărul **suma**. Altfel spus, soluția se obține atunci când suma $s=a[st[1]]+a[st[2]]+\dots+a[st[k-1]]+a[st[k]]$ are valoarea **suma**.

- **Condiția de continuare** este ca suma valorilor termenilor generați, s , să fie mai mică sau egală cu suma ($s \leq \text{suma}$), o soluție completă obținându-se atunci când $s = \text{suma}$. Atunci când se urcă în stivă, se adună la sumă valoarea bancnotei cu indicele k : $s += a[\text{st}[k]]$, iar când se coboară în stivă, se scade din sumă valoarea bancnotei cu indicele k : $s -= a[\text{st}[k]]$ și valoarea bancnotei cu indicele $k-1$: $s += a[\text{st}[k-1]]$.
- Deoarece este posibil ca – pentru anumite valori ale sumei suma și ale valorilor bancnotelor – să nu existe nici o soluție, se folosește variabila de tip logic **este**, care are valoarea 1 (*true*), dacă există cel puțin o soluție, și 0 (*false*), în caz contrar.

```
#include<iostream.h>
typedef int stiva[100];
int n,k,ev,as,s=0,suma,este=0,a[10];
stiva st;
void init ()
{if (k==1) st[k]=0;
 else st[k]=st[k-1]-1;}
int succesor()
{if (st[k]<n) {st[k]=st[k]+1; return 1;}
 else {s-=a[st[k-1]]; return 0;}}
int valid()
{if (s+a[st[k]]<=suma) {s+=a[st[k]]; return 1;}
 else return 0;}
int solutie()
{return s==suma;}
void tipar()
{int i,j,p,este=1;
 for (i=1;i<=n;i++)
 {for(j=1,p=0;j<=k;j++) if (i==st[j]) p++;
 if (p!=0) cout<<p<<"* "<<a[i]<<" + ";
 cout<<'\'b'<<'\'b'<<" " <<endl; s-=a[st[k]];}
void bt() {//partea fixă a algoritmului }
void main()
{cout<<"n= "; cin>>n;
 for (int i=1;i<=n;i++)
 {cout<<"Valoare bancnota "<<i<<": "; cin>>a[i];
 cout<<"suma= "; cin>>suma; bt();
 if (!este) cout<<"Imposibil"; }}
```

Temă

Scriți următoarele programe, în care să folosiți metoda backtracking pentru generarea tuturor partițiilor unui număr natural.

1. Să se genereze toate descompunerile unui număr natural n în numere naturale distincte.
2. Să se genereze toate descompunerile unui număr natural n în numere prime.
3. Să se genereze toate descompunerile unui număr natural n în sumă de 3 și 5.
4. O bară are lungimea L . Se consideră n reper de lungimi diferite. Să se genereze toate posibilitățile de a tăia bara după reperele existente, fără să rămână rest la tăiere, un reper putând fi folosit de mai multe ori. Se poate ca unele repere să nu fie folosite. Se citesc dintr-un fișier text, de pe primul rând, lungimea barei – L și numărul de repere – n , iar de pe următorul rând, reperele. Numerele de pe un rând sunt separate prin spațiu.

- Pentru realizarea unui chestionar există n întrebări, fiecare întrebare având un punctaj. Numărul de întrebări și punctajul fiecărei întrebări se citesc dintr-un fișier text. Să se genereze toate chestionarele care au între a și b întrebări distincte și un punctaj total între p și q puncte. Valorile pentru a , b , p și q se citesc de la tastatură.
- Să se găsească modalitatea de plată a unei sume cu un număr minim de bancnote cu valori date.

1.3.3.6. Generarea tuturor partițiilor unei multimi

O partitie a unei multimi A este formată din multimile nevide disjuncte A_i a căror reuniune este multimea A : $\bigcup_{i=1}^m A_i = A$ și $A_i \cap A_j = \emptyset$ pentru orice $i, j = 1 \dots m$.

Pentru simplificarea algoritmului vom considera multimea $A = \{1, 2, 3, \dots, n\}$. O partitie va fi formată din m multimi, cu $1 \leq m \leq n$. Soluția va fi memorată în stivă și va avea n elemente, fiecare element k al soluției reprezentând multimea i ($1 \leq i \leq n$) căreia îi aparține elementul k din multimea care se partionează: $st[k] = i$ înseamnă că elementul k din multimea A face parte din multimea i a partitiei. În cadrul unei partitii nu interesează ordinea în care apar elementele multimii A . Cel mai mare număr care va fi generat în stivă reprezintă numărul de multimi m în care a fost descompusă multimea A . În plus, numerele generate în stivă trebuie să aparțină unei multimi de numere consecutive care începe cu 1, deoarece partitia nu conține multimi vide. Alăturat, sunt prezentate toate partitiiile multimii $\{1, 2, 3\}$ și conținutul stivei pentru fiecare dintre ele. În exemplu, nu există soluția 1 3 3 deoarece aceste valori nu aparțin unei multimi de numere consecutive (nu se poate ca un element din multimea A să aparțină multimii A_1 , și alte două elemente multimii A_3 , deoarece ar însemna că multimea A_2 este vidă).

Partitiiile	Stiva
{1, 2, 3}	1 1 1
{1, 2} {3}	1 1 2
{1, 3} {2}	1 2 1
{1} {2, 3}	1 2 2
{1} {2} {3}	1 2 3

Condiția de continuare este asigurată prin modul în care este ales succesorul $st[k] \leq k$, ceea ce înseamnă că elementul k din multimea A nu poate aparține decât unei partitii al cărei număr este mai mic sau cel mult egal cu numărul de ordine al elementului. Altfel spus, dacă până la nivelul k numărul maxim atribuit pentru o partitie este i , acest număr reprezentând numărul de multimi ale partitiei care există în soluția parțială, pe nivelul $k+1$ elementul poate avea una dintre următoarele valori:

- Orice valoare de la 1 la i , ceea ce înseamnă că elementul $k+1$ din multimea A se adaugă la una dintre multimile care există deja.
- Valoarea $i+1$, ceea ce înseamnă că elementul $k+1$ din multimea A va genera o nouă multime în partitie.

Implementarea iterativă

```
#include<iostream.h>
typedef int stiva[100];
int n,k,ev,as;
stiva st;
void init() {st[k]=0;}
int succesor()
{if (st[k]<st[k-1]+1)
    {st[k]=st[k]+1; return 1;}
 else return 0;}
```

Implementarea recursivă

```
#include<iostream.h>
typedef int stiva[100];
int n;
stiva st;
void init (int k) {st[k]=0;}
int succesor(int k)
{if (st[k]<st[k-1]+1)
    {st[k]=st[k]+1;return 1;}
 else return 0;}
```

```

int valid() {return 1;}
int solutie()
{return k==n;}
void tipar()
{int i,j,max=st[1];
for(i=2;i<=n;i++)
  if (st[i]>max) max=st[i];
for (i=1;i<=max;i++)
  {cout<<"'";
   for (j=1;j<=n;j++)
     if (st[j]==i) cout<<j<<",";
   cout<<'\'b'<<"")  ";}
cout<<endl;}
void bt()
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n; bt();}

```

```

int valid() {return 1;}
int solutie(int k)
{return k==n;}
void tipar()
{int i,j,max=st[1];
for(i=2;i<=n;i++)
  if (st[i]>max) max=st[i];
for (i=1;i<=max;i++)
  {cout<<"'";
   for (j=1;j<=n;j++)
     if (st[j]==i) cout<<j<<",";
   cout<<'\'b'<<"")  ";}
cout<<endl;}
void bt(int k)
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n; bt(1);}

```

Temă

Scrieți următoarele programe, în care să folosiți metoda backtracking pentru generarea tuturor partițiilor unei mulțimi. Se consideră mulțimea A, cu n numere întregi. Valorile pentru n și m și pentru elementele mulțimii A se citesc de la tastatură.

1. Să se genereze toate partițiile mulțimii A formate din două submulțimi care au suma elementelor egale.
2. Să se genereze toate partițiile mulțimii A formate din m submulțimi.
3. Să se genereze toate partițiile mulțimii A formate din submulțimi care au același număr de elemente.

1.3.3.7. Generarea tuturor funcțiilor surjective

Se generează toate funcțiile surjective $f:A \rightarrow B$ unde $\text{card}(A)=m$ și $\text{card}(B)=n$. Pentru simplificarea algoritmului vom considera mulțimile $A=\{1,2,3,\dots,m\}$ și $B=\{1,2,3,\dots,n\}$. O soluție este formată din m elemente. Elementul k al soluției reprezintă valoarea funcției: $f(k)$. Deoarece valoarea funcției $f(k)$ aparține mulțimii B, în stivă se vor genera numere din mulțimea $\{1,2,3,\dots,n\}$. Din definiția funcției surjective, trebuie ca, pentru orice $j \in B$, să existe $i \in A$, astfel încât $f(i)=j$. Rezultă că – pentru ca funcția să fie surjectivă – trebuie ca $n \leq m$. Problema se reduce la generarea în stivă a tuturor elementelor produsului cartezian B^m din care vor fi considerate soluții numai cele care conțin toate elementele din mulțimea B. Soluțiile vor fi afișate sub forma tabelului de variație a funcției. De exemplu, dacă $A=\{1,2,3\}$ și $B=\{1,2\}$, soluțiile și modul în care vor fi afișate sunt prezentate alăturat.

Condiția de continuare este asigurată prin modul în care este ales succesorul ca element din mulțimea B, singurul caz special fiind al elementului care se adaugă pe ultimul nivel din stivă (m). Prin adăugarea acestui element, trebuie ca în stivă să existe toate elementele

	Soluția	Afișarea
1 1 2	x 1 2 3 f(x) 1 1 2	
1 2 1	x 1 2 3 f(x) 1 2 1	
1 2 2	x 1 2 3 f(x) 1 2 2	
2 1 1	x 1 2 3 f(x) 2 1 1	
2 2 1	x 1 2 3 f(x) 2 2 1	
2 1 2	x 1 2 3 f(x) 2 1 2	
2 2 1	x 1 2 3 f(x) 2 2 1	

mulțimii B. Această condiție este verificată prin funcția `surjectiva()` care furnizează un rezultat logic: 1 (`true`), dacă în stivă există toate cele n elemente ale mulțimii B, și 0 (`false`), în caz contrar.

Implementarea iterativă

```
#include<iostream.h>
typedef int stiva[100];
int n,m,k,ev,as;
stiva st;
int surjectiva()
{int i,j,x;
 for (j=1;j<=n;j++)
 {for (i=1,x=0;i<=m && !x;i++)
   if (st[i]==j) x=1;
 if (!x) return 0;}
return 1;}
void init() {st[k]=0;}
int succesor()
{if (st[k]<st[k-1]+1)
 {st[k]=st[k]+1; return 1;}
 else return 0;}
int valid()
{if (k==m)
  if (!surjectiva()) return 0;
return 1;}
int solutie()
{return k==m;}
void tipar()
{int i; cout<<" x | ";
 for (i=1;i<=m;i++) cout<<i<<" ";
 cout<<endl;
 for (i=1;i<=m;i++) cout<<"----";
 cout<<endl<<"f(x) | ";
 for (i=1;i<=m;i++) cout<<st[i]<<" ";
 cout<<endl<<endl;}
void bt()
{//partea fixă a algoritmului}
void main()
{cout<<"elemente multimea A= ";
 cin>>m;
 cout<<"elemente multimea B= ";
 cin>>n; bt();}
```

Implementarea recursivă

```
#include<iostream.h>
typedef int stiva[100];
int n,m;
stiva st;
int surjectiva()
{int i,j,x;
 for (j=1;j<=n;j++)
 {for (i=1,x=0;i<=m && !x;i++)
   if (st[i]==j) x=1;
 if (!x) return 0;}
return 1;}
void init (int k) {st[k]=0;}
int succesor(int k)
{if (st[k]<st[k-1]+1)
 {st[k]=st[k]+1;return 1;}
 else return 0;}
int valid(int k)
{if (k==m)
  if (!surjectiva()) return 0;
return 1;}
int solutie(int k)
{return k==n;}
void tipar()
{int i; cout<<" x | ";
 for (i=1;i<=m;i++) cout<<i<<" ";
 cout<<endl;
 for (i=1;i<=m;i++) cout<<"----";
 cout<<endl<<"f(x) | ";
 for (i=1;i<=m;i++) cout<<st[i]<<" ";
 cout<<endl<<endl;}
void bt(int k)
{//partea fixă a algoritmului}
void main()
{cout<<"elemente multimea A= ";
 cin>>m;
 cout<<"elemente multimea B= ";
 cin>>n; bt(1);}
```


Temă

Scrieți următoarele programe, în care să folosiți metoda backtracking pentru generarea tuturor funcțiilor surjective.

1. Se citesc de la tastatură n cifre distincte. Să se genereze toate numerele de m cifre ($n \leq m$) care se pot forma cu aceste cifre și care conțin toate cele n cifre.
2. Se citesc de la tastatură n caractere distincte. Să se genereze toate cuvintele de m caractere ($n \leq m$) care se pot forma cu aceste caractere și care conțin toate cele n caractere.

3. Profesorul de informatică a pregătit m teme, pentru proiecte pe care trebuie să le repartizeze celor n elevi din clasă ($m \leq n$), astfel încât nici o temă de proiect să nu rămână nerepartizată. Să se genereze toate soluțiile de repartizare a temelor pentru proiect.
4. O bară are lungimea L . Se consideră n reperete de lungimi diferite. Să se genereze toate posibilitățile de a tăia bara după reperete existente, fără să rămână rest la tăiere, fiecare reper fiind folosit cel puțin o dată. Se citesc dintr-un fișier text, de pe primul rând, lungimea barei – L și numărul de reperete – n , iar de pe următorul rând, reperetele. Numerele de pe un rând sunt separate prin spațiu.
5. Să se genereze toate construcțiile corecte de paranteze (si); n este număr par și se citește de la tastatură. De exemplu, pentru $n=6$, construcțiile $((())()$ și $()((()$ sunt corecte, iar construcția $)()()$ nu este corectă. **Indicație.** Se generează funcțiile surjective $f:A \rightarrow B$ unde $A = \{1,2,3,\dots,n\} =$ mulțimea pozițiilor ocupate de paranteze, și $B = \{0,1\} =$ mulțimea parantezelor – ($=0$ și $=1$) – care îndeplinesc următoarele condiții:
- $f(1)=0$ și $f(n)=1$ (expresia începe cu paranteză deschisă și se termină cu o paranteză închisă).
 - Numărul de valori 0 ale funcției și numărul de valori 1 ale funcției sunt egale cu $n/2$ (numărul de paranteze deschise este egal cu numărul de paranteze închise).
 - În timpul construirii soluției, nu trebuie ca numărul de valori 1 ale funcției să fie mai mare decât numărul de valori 0 generate până la acel moment (numărul de paranteze închise este întotdeauna cel mult egal cu numărul de paranteze deschise).

1.3.3.8. Problema celor n dame

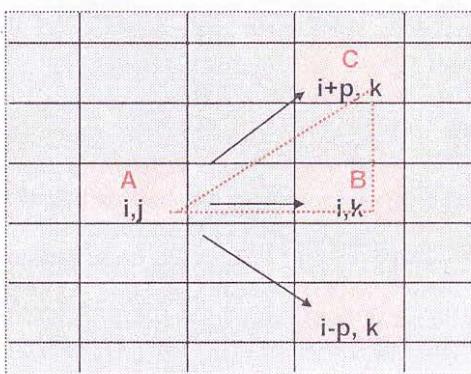
Se generează toate posibilitățile de aranjare, pe o tablă de șah, cu dimensiunea $n \times n$, a n dame care să nu se atace între ele. Damele se pot ataca între ele pe linie, pe coloană și pe diagonală.

Se observă că fiecare damă trebuie să fie plasată singură pe o coloană, ca să nu se atace între ele. Soluția problemei este data de mulțimea cu n elemente $\{x_1, x_2, \dots, x_n\}$ care se memorează în stivă. Elementul soluției x_k reprezintă numărul liniei în care se așază dama din coloana k , și se memorează pe nivelul k al stivei st . De exemplu, pentru $n=8$, o soluție este $S=\{4,8,1,5,7,2,6,3\}$ și damele sunt aranjate pe tabla de șah ca în figura alăturată.

	8		◎						
	7				◎				
	6					◎			
	5				◎				
	4	◎							
	3						◎		
	2						◎		
	1		◎						
	1	2	3	4	5	6	7	8	

Date fiind coordonatele i,j , de pe tabla de șah ale poziției unei dame care a fost așezată anterior (linia i este memorată pe nivelul j în stivă – $i=st[j]$), ca ea să nu atace dama care urmează să fie pusă în coloana k – trebuie să fie îndeplinite următoarele **condiții interne**:

- Dama din coloana k nu trebuie să se găsească pe linia i .
- Dama din coloana k nu trebuie să se găsească pe diagonală cu dama din coloana j , adică triunghiul ABC nu trebuie să fie isoscel. Pentru ca triunghiul ABC să



nu fie isoscel, condiția pe care trebuie să o îndeplinească dama din coloana k se poate exprima astfel: oricare ar fi $j < k$, $x_j \neq x_k$ și $|x_k - x_j| \neq k - j$.

Aceste condiții interne ale soluției trebuie să se regăsească în condiția de continuare a soluției – care se verifică în subprogramul `valid()`: $st[j] \neq st[k]$ și $\text{abs}(st[k] - st[j]) \neq k - j$, pentru orice $j \neq k$.

```
#include<iostream.h>
#include<math.h>
typedef int stiva[100];
int n,k,ev,as;
stiva st;
void init()
{st[k]=0;}
int succesor()
{if (st[k]<n) {st[k]=st[k]+1; return 1;}
 else return 0;}
int valid()
{for(int i=1;i<k;i++)
 if (st[k]==st[i] || abs(st[k]-st[i])==k-i) return 0;
 return 1;}
int solutie()
{return k==n;}
void tipar()
{for(int i=1;i<=n;i++) cout<<st[i]<<" ";
 cout<<endl;}
void bt()
{//partea fixă a algoritmului }
void main()
{cout<<"n= "; cin>>n; bt();}
```



Tema

Să se genereze toate posibilitățile de aranjare, pe o tablă de șah, cu dimensiunea $n \times n$, a n nebuni, care să nu se atace între ei. Nebunul se poate deplasa numai pe diagonală.

1.3.3.9. Parcurea tablei de șah cu un cal

Se caută toate posibilitățile de parcuregere a tablei de șah prin săritura calului, fără a trece de două ori prin aceeași poziție. Tabla de șah are dimensiunea $n \times n$, iar poziția inițială a calului este dată de i – numărul liniei și j – numărul coloanei. Se citesc de la tastatură valorile pentru n , i și j .

Soluția problemei este dată de mulțimea cu $n \times n$ elemente $\{x_1, x_2, \dots, x_{n \times n}\}$ care se memorează în stivă. Elementul soluției x_k reprezintă coordonatele i și j ale pătratului în care se deplasează calul la mutarea k – și se memorează pe nivelul k al stivei `st`. Pentru a înregistra traseul străbătut de cal, elementele stivei vor fi de tip înregistrare cu două câmpuri, x și y , corespunzătoare celor două coordonate ale pătratului.

```
struct element{int x,y;};
typedef element stiva[100];
stiva st;
```

Stiva va avea dimensiunea $n \times n$, corespunzătoare parcurgerii întregii table de șah ($k_{\max} = n \times n$), și va conține, în ordine, coordonatele tuturor pătratelor de pe traseul de parcuregere: `st[1].x` și `st[1].y` corespund primei poziții de pe tablă, `st[2].x` și `st[2].y` corespund celei de a doua poziții de pe tablă, ..., `st[n*n].x` și `st[n*n].y` corespund ultimei poziții de pe tablă.

De exemplu, pentru $n=5$ – și ca poziție de pornire pătratul de pe linia 1 și coloana 1 – o soluție va avea 25 de elemente. Prima dintre soluțiile obținute este prezentată mai jos:

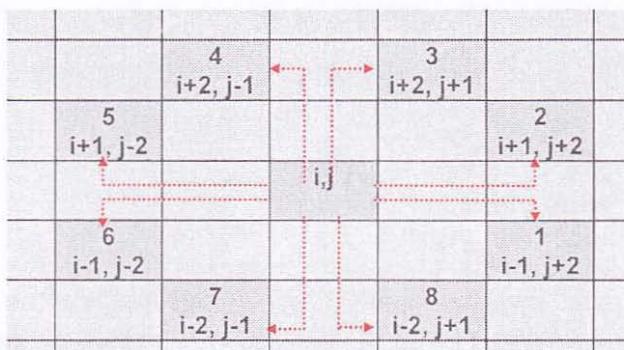
Soluția				
1,1	3,2	5,1	4,3	5,5
3,4	4,2	5,4	3,5	1,4
2,2	4,1	5,3	4,5	2,4
1,2	3,1	5,2	3,3	2,1
1,3	2,5	4,4	2,3	1,5

Ordinea de ocupare a tablării de șah

1	1	16	21	10	25
2	20	11	24	15	22
3	17	2	19	6	9
4	12	7	4	23	14
5	3	18	13	8	5

Dată fiind o poziție curentă i,j a traseului, variantele posibile pentru următoarea poziție a calului pe tablă de șah sunt prezentate alăturat.

Se observă că există 8 variante posibile pentru continuarea drumului. Coordonatele următorului pătrat de pe tablă de șah în care poate sări calul, pentru fiecare dintre cele 8 variante, sunt prezentate în tabel.



Variantă	Coordonate	Variantă	Coordonate
1	$i-1, j+2$	5	$i+1, j-2$
2	$i+1, j+2$	6	$i-1, j-2$
3	$i+2, j+1$	7	$i-2, j-1$
4	$i+2, j-1$	8	$i-2, j+1$

În vectorul p , de dimensiune $n \times n$ se păstrează varianta care s-a ales pentru continuarea drumului. Elementele sale sunt de tip întreg. Elementul k din vector arată care dintre cele 8 variante s-a ales

pentru a ajunge de la poziția $k-1$ la poziția k . Orice element k al vectorului trebuie să îndeplinească următoarea condiție: $1 \leq p[k] \leq 8$.

Fiecare dintre cele 8 variante de deplasare din pătratul $k-1$ în pătratul k înseamnă adăugarea unei constante de deplasare la coordonata x , respectiv y , a pătratului $k-1$. Aceste constante sunt memorate în vectorul d (pentru varianta 0, care nu există, deplasarea este 0):

element $d[9]=\{(0,0), (-1,2), (1,2), (2,1), (2,-1), (1,-2), (-1,-2), (-2,-1), (-2,1)\}$;

Variabila logică este se folosește pentru a ști dacă s-a găsit cel puțin o soluție (de exemplu, pentru $n=4$ problema nu are soluții). Este inițializată cu valoarea 0 (nu s-a găsit încă nici o soluție). Va lua valoarea 1 (s-a găsit o soluție) în subprogramul `tipar()` care se execută numai dacă s-a găsit o soluție a problemei.

Subprogramele

→ Subprogramul `init()`. Se inițializează varianta cu care se trece din pătratul $k-1$ în pătratul k , adică elementul k din vectorul p : $p[k]=0$.

- Subprogramul **succesor()**. Se consideră că mai există o posibilitate de continuare a construirii soluției pe nivelul **k** dacă mai există o variantă de mutare din pătratul **k-1** în pătratul **k**, adică dacă $p[k] < 8$. Dacă mai există o variantă de mutare, pe nivelul **k** din stivă, se trece la această variantă ($p[k]=p[k]+1$) și se înregistrează coordonatele pătratului **k**, care se vor obține prin adăugarea, la coordonatele pătratului **k-1**, a deplasărilor corespunzătoare acestei variante ($st[k].x=st[k-1].x+d[p[k]].x$ și $st[k].y=st[k-1].y+d[p[k]].y$).
- Subprogramul **valid()**. Se consideră că pătratul **k** în care s-a ajuns poate fi considerat că este bun pentru a continua construirea soluției, dacă coordonatele sale nu sunt în afara tablei de șah (coordonatele $st[k].x$ și $st[k].y$ iau valori în intervalul **[1,n]**) și prin pătratul **k** nu s-a mai trecut (se verifică dacă în stivă, pe nivelurile anterioare, nu mai există un pătrat cu coordonatele pătratului **k**).
- Subprogramul **soluție()**. Se verifică dacă a fost parcursă toată tabla de șah, adică dacă vârful stivei **k** are valoarea **n*n**.
- Subprogramul **bt()**. Deoarece coordonatele primului pătrat nu trebuie modificate, primului element al soluției (**k=1**) î se atribuie coordonatele în subprogramul **main()**, iar în subprogram se va începe cu initializarea nivelului 2 din stivă (**k=2**); căutarea tuturor soluțiilor se va face până când vârful stivei coboară până la primul element al soluției: **while (k>1)**.

```
#include<iostream.h>
struct element{int x,y;};
element d[9]={{0,0}, {-1,2}, {1,2}, {2,1}, {2,-1}, {1,-2}, {-1,-2}, {-2,-1}, {-2,1}};
int n,k,ev,as,p[100],este=0;
typedef element stiva[100];
stiva st;
void init() {p[k]=0;}
int succesor()
{if (p[k]<8)
 {p[k]=p[k]+1; st[k].x=st[k-1].x+d[p[k]].x;
  st[k].y=st[k-1].y+d[p[k]].y; return 1;}
 else return 0;}
int valid()
{if (st[k].x<1 || st[k].y<1 || st[k].x>n || st[k].y>n) return 0;
 for (int i=1;i<k;i++)
  if (st[k].x==st[i].x && st[k].y==st[i].y) return 0;
 return 1;}
int solutie() {return k==n*n;}
void tipar()
{int i; este=1;
 for (i=1;i<=n*n;i++) cout<<st[i].x<<","<<st[i].y<<" ";
 cout<<endl;}
void bt()
{k=2; init();
 while (k>1)
 {as=1; ev=0;
  while(as && !ev)
   {as=succesor();
    if(as) ev=valid();}}
```

```

if(as)
    if(solutie()) tipar();
    else {k++; init();}
else k--;}
void main()
{int i,j; cout<<"n= "; cin>>n;
cout<<"linia de pornire "; cin>>i; st[1].x=i;
cout<<"coloana de pornire "; cin>>j; st[1].y=j; bt();
if (!este) cout<<"Nu există soluții";}

```

Observație. Dacă se dorește **numai afișarea primei soluții**, căutarea soluției se face până când variabila **este** trece de la valoarea 0 (nu s-a găsit soluție) la valoarea 1 (s-a găsit soluție), modificându-se în subprogramul **bt()** condiția de terminare a căutării tuturor soluțiilor (în loc de **while (k>1)** se va scrie **while (k>1 && !este)**).

Temă


Să se genereze toate posibilitățile de aranjare pe o tablă de șah, cu dimensiunea **n**×**n**, a **n** cai care să nu se atace între ei.

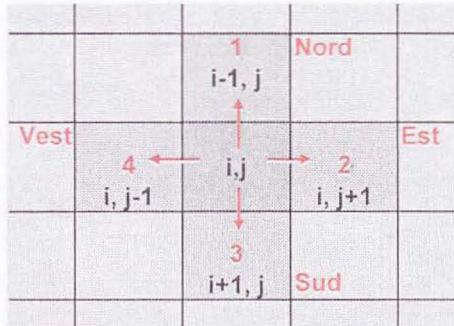
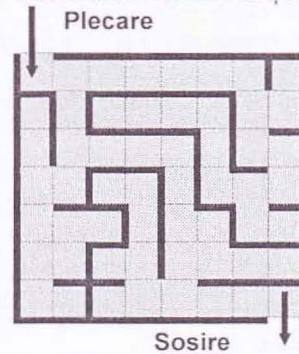
1.3.3.10. Generarea tuturor soluțiilor de ieșire din labirint

Se caută toate posibilitățile unui traseu printr-un labirint, între două poziții, inițială și finală, fără a se trece de două ori prin aceeași poziție. Labirintul are dimensiunea **n**×**m**. Poziția inițială este dată de **x₁** – numărul liniei – și **y₁** – numărul coloanei – care se comunică de la tastatură. Poziția finală corespunzătoare ieșirii din labirint este dată de **x₂** – numărul liniei – și **y₂** – numărul coloanei – care vor fi determinate în urma găsirii traseului. Deplasarea în labirint se poate face numai ortogonal (pe verticală și orizontală), nu și pe diagonală.

Soluția problemei este dată de mulțimea cu **t** elemente $\{x_1, x_2, \dots, x_t\}$ care se memorează în stivă. Numărul de elemente ale soluției depinde de soluția găsită. Elementul soluției **x_k** reprezintă coordonatele **i** și **j** ale pătratului în care se deplasează o persoană în labirint și se memorează pe nivelul **k** al stivei **st**.

Pentru a înregistra traseul străbătut, elementele stivei vor fi de tip înregistrare cu două câmpuri, **x** și **y**, corespunzătoare celor două coordonate ale pătratului. Stiva va avea dimensiunea **t** corespunzătoare parcurgerii întregului traseu, de la pătratul de plecare până la pătratul de sosire, și va conține, în ordine, coordonatele tuturor păratelor de pe traseul de parcurgere: **st[1].x** și **st[1].y** corespund păratului de plecare, **st[2].x** și **st[2].y** corespund celui de al doilea părat în care se trece, ..., **st[t].x** și **st[t].y** corespund păratului de sosire, prin care se ieșe din labirint.

Dată fiind o poziție curentă **i,j** a traseului, variantele posibile pentru următoarea poziție sunt prezentate alăturat.



Se observă că există 4 variante posibile pentru continuarea drumului. Coordonatele următorului pătrat de pe traseu în care poate să treacă persoana, pentru fiecare dintre cele 8 variante, sunt prezentate în tabel.

Variantă	Coordonate
1	i+1, j
2	i, j+1
3	i+1, j
4	i, j-1

În vectorul p , de dimensiune $n \times m$, se păstrează varianta care s-a ales pentru continuarea drumului. Elementele sale sunt de tip întreg. Elementul k din vector arată care dintre cele 4 variante s-a ales pentru a ajunge de la poziția $k-1$ la poziția k . Orice element k al vectorului trebuie să îndeplinească următoarea condiție: $1 \leq p[k] \leq 4$.

Fiecare dintre cele 4 variante de deplasare din pătratul $k-1$ în pătratul k înseamnă adăugarea unei constante de deplasare la coordonata x , respectiv y , a pătratului $k-1$. Aceste constante sunt memorate în vectorul d (pentru varianta 0, care nu există, deplasarea este 0):

```
element d[5]={{0,0}, {-1,0}, {0,1}, {1,0}, {0,-1}};
```

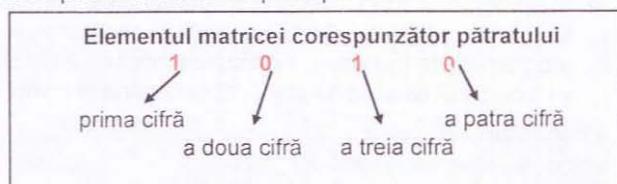
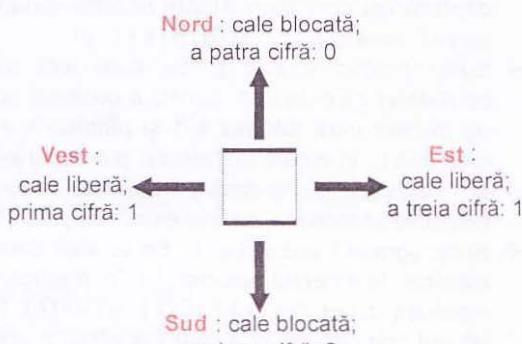
Deoarece din pătratul $k-1$ se poate trece în pătratul k , găsit pe una dintre cele patru direcții de deplasare numai dacă cele două pătrate comunică (există un culoar între ele) trebuie găsită o metodă de descriere a labirintului printr-o matrice L cu dimensiunea $n \times m$. Vor fi prezentate două metode de reprezentare a labirintului.

Varianta 1

Fiecare element al matricei L corespunde unui pătrat din labirint și memorează informații despre mișările care pot fi executate în pătrat. Fiecarui pătrat i se atribuie un sir de patru cifre binare, care va fi construit după următoarele reguli:

- prima cifră: are valoarea 1, dacă se poate executa un pas spre Vest;
- a doua cifră: are valoarea 1, dacă se poate executa un pas spre Sud;
- a treia cifră: are valoarea 1, dacă se poate executa un pas spre Est;
- a patra cifră: are valoarea 1, dacă se poate executa un pas spre Nord.

Pentru pătratul din exemplu este prezentată alăturat valoarea elementului care i se asociază în matricea L . Sirul de patru cifre binare formează un număr binar, care va fi transformat într-un număr zecimal.



0011	1110	1010	1010	1010	1010	1100	0100
0100	0101	0010	1010	1010	1100	0111	1001
0101	0111	1010	1010	1100	0101	0011	1100
0111	1001	0010	1100	0101	0011	1110	1001
0111	1110	1000	0101	0111	1100	0011	1000
0111	1110	1000	0101	0111	1100	0011	1000
0111	1001	0010	1101	0111	1011	1010	1000
0011	1000	0010	1011	1011	1010	1010	1100

3	14	10	10	10	10	12	4
4	5	2	10	10	12	7	9
5	7	10	10	12	5	3	12
7	9	2	12	5	3	14	9
7	14	8	5	7	10	3	8
7	9	2	13	7	11	10	8
3	8	2	11	11	10	10	12

Pentru a simplifica verificarea condiției de ieșire din labirint, matricea L este bordată cu o valoare pe care nu o poate lua nici unul dintre elementele ei (de exemplu, valoarea 16). Matricea L are 7 linii și 8 coloane.

Soluția obținută va afișa coordonatele pătratelor prin care se trece:

Soluția				
1,1	1,2	2,2	3,2	3,3
3,4	3,5	4,5	5,5	6,5
7,5	7,6	7,7	7,8	

1	1	2					
2		3					
3		4	5	6	7		
4					8		
5					9		
6					10		
7					11	12	13
					14		

Subprogramele

- Subprogramul `init()`. Se inițializează varianta cu care se trece din pătratul $k-1$ în pătratul k , adică elementul k din vectorul p : $p[k]=0$.
- Subprogramul `succesor()`. Se consideră că mai există o posibilitate de continuare a construirii soluției pe nivelul k dacă mai există o variantă de deplasare din pătratul $k-1$ în pătratul k , adică dacă $p[k] < 4$. Dacă mai există o variantă de deplasare, pe nivelul k din stivă, se trece la această variantă ($p[k]=p[k]+1$) și se înregistrează coordonatele pătratului k , care se vor obține prin adăugarea, la coordonatele pătratului $k-1$, a deplasărilor corespunzătoare acestei variante ($st[k].x=st[k-1].x+d[p[k]].x$ și $st[k].y=st[k-1].y+d[p[k]].y$).
- Subprogramul `valid()`. Se consideră că pătratul k , în care s-a ajuns, poate fi considerat că este bun, pentru a continua construirea soluției, numai dacă există culoar de trecere între pătratul $k-1$ și pătratul k – pe direcția de deplasare aleasă (dacă în matricea L, în elementul asociat pătratului $k-1$, bitul corespunzător direcției de deplasare are valoarea 1) – și dacă prin pătratul k nu s-a mai trecut (se verifică dacă în stivă, pe nivelurile anterioare, nu mai există un pătrat cu coordonatele pătratului k).
- Subprogramul `soluție()`. Se verifică dacă pătratul în care s-a ajuns se află în afara matricei (elementul asociat lui în matricea L are valoarea cu care a fost bordată matricea: $L[st[k].x][st[k].y]==16$). Pentru a evita soluția prin care seiese din labirint prin pătratul prin care s-a intrat, în stivă trebuie să existe cel puțin două elemente ($k>2$). Deoarece ultimul pătrat din stivă nu face parte din soluția problemei (este ca un terminator al soluției), coordonatele lui nu vor fi afișate în subprogramul `tipar()`.
- Subprogramul `bt()`. Deoarece coordonatele primului pătrat nu trebuie modificate, primul element al soluției ($k=1$) î se atribuie coordonatele în subprogramul `main()`, iar în subprogram se va începe cu inițializarea nivelului 2 din stivă ($k=2$); căutarea tuturor soluțiilor se va face până când vârful stivei coboară până la primul element al soluției: `while (k>1)`.

```
#include<fstream.h>
struct element{int x,y;};
element d[5]={{0,0},{-1,0},{0,1},{1,0},{0,-1}};
int n,m,k,ev,as,p[100],este=0,L[10][10];
typedef element stiva[100];
stiva st;
fstream f("labirint1.txt",ios::in);
void init() {p[k]=0;}
int successor()
{
if (p[k]<4) {p[k]=p[k]+1;st[k].x=st[k-1].x+d[p[k]].x;
st[k].y=st[k-1].y+d[p[k]].y; return 1;}
}
```

```

    else return 0;}
int valid()
{int x=st[k-1].x,y=st[k-1].y;
for (int i=1;i<k;i++)
    if (st[k].x==st[i].x && st[k].y==st[i].y) return 0;
switch (p[k])
    {case 1: if (L[x][y] & 1) return 1; break; //spre N
     case 2: if (L[x][y] & 2) return 1; break; //spre E
     case 3: if (L[x][y] & 4) return 1; break; //spre S
     case 4: if (L[x][y] & 8) return 1; }           //spre V
return 0;}
int solutie() {return k>2 && L[st[k].x][st[k].y]==16;}
void tipar()
{int i; este=1;
for (i=1;i<k;i++) cout<<st[i].x<<","<<st[i].y<<" ";
cout<<endl;}
void bt()//partea fixă a algoritmului ca în exemplul anterior}
void main()
{int i,j; f>>n>>m;
for (i=1;i<=n;i++)
    for (j=1;j<=m;j++) f>>L[i][j];
cout<<"Intrarea în labirint: "<<endl;
cout<<"x= "; cin>>i; st[1].x=i; cout<<"y= "; cin>>j; st[1].y=j;
for (i=1;i<=n;i++) {L[i][0]=16; L[i][m+1]=16;}
for (i=1;i<=m;i++) {L[0][i]=16; L[n+1][i]=16;}
bt(); if (!este) cout<<"Nu există soluții";}

```

Varianta 2

Fiecare element al matricei L corespunde unei pozitii din labirint.

$$L(i,j) = \begin{cases} 1, & \text{dacă se poate trece prin poziția } (i,j) - \text{este culoar} \\ 0, & \text{dacă nu se poate trece prin poziția } (i,j) - \text{este zid} \end{cases}$$

Matricea L are 15 linii și 17 coloane. Punctul de intrare în labirint se găsește pe linia 1 și în coloana 2.

A large grid of squares, mostly empty, with some squares filled with dots. The grid consists of approximately 20 columns and 15 rows. The filled squares are located in various positions, including a cluster in the lower-left quadrant, a single square in the middle-right area, and several more scattered throughout the grid.

Soluția obținută va afișa coordonatele punctelor prin care se trece:

Soluția				
1,2	2,2	2,3	2,4	3,4
4,4	5,4	6,4	6,5	6,6
6,7	6,8	6,9	6,10	7,10
8,10	9,10	10,10	10,11	11,11
12,11	12,10	13,10	14,10	14,11
14,12	14,13	14,14	14,15	14,16
15,16				

Subprogramele (sunt prezentate numai modificările față de varianta anterioară):

→ Subprogramul **valid()**. Se consideră că poziția **k**, în care s-a ajuns, poate fi considerată că este bună, pentru a continua construirea soluției, numai dacă ea este corectă și dacă prin poziția **k** nu s-a mai trecut (se săsteacă că nu mai există o pozitie cu coordonatele po-

→ Subprogramul **soluție()**. Se verifică dacă poziția în care s-a ajuns nu se află la ieșirea din labirint, adică se găsește pe marginea matricei ($st[k].x==1$ sau $st[k].x==n$ sau $st[k].y==1$ sau $st[k].y==m$).

```

#include<fstream.h>
struct element{int x,y;};
element d[5]={{0,0}, {-1,0}, {0,1}, {1,0}, {0,-1}};
int n,m,k,ev,as,p[100],este=0,L[10][10];
typedef element stiva[100];
stiva st;
fstream f("labirint1.txt",ios::in);
void init() {p[k]=0;}
int succesor()
{if (p[k]<4) {p[k]=p[k]+1;st[k].x=st[k-1].x+d[p[k]].x;
               st[k].y=st[k-1].y+d[p[k]].y; return 1;}
  else return 0;}
int valid ()
{if (L[st[k].x][st[k].y]==0) return 0;
 for (int i=1;i<k;i++)
   if (st[k].x==st[i].x && st[k].y==st[i].y) return 0;
 return 1;}
int solutie()
{return st[k].x==1 || st[k].x==n || st[k].y==1 || st[k].y==m;}
void tipar()
{for (int i=1,este=1;i<=k;i++) cout<<st[i].x<<","<<st[i].y<<" ";
 cout<<endl;}
void bt()//partea fixă a algoritmului că în exemplul anterior)
void main()
{int i,j; f>>n>>m;
 for (i=1;i<=n;i++)
   for (j=1;j<=m;j++) f>>L[i][j];

```

```

cout<<"Intrarea in labirint: "<<endl;
cout<<"x= "; cin>>i; st[1].x=i; cout<<"y= "; cin>>j; st[1].y=j;
bt(); if (!este) cout<<"Nu exista solutii";)

```

Observație. Algoritmul folosit pentru generarea tuturor traseelor de ieșire dintr-un labirint poate fi folosit în orice problemă de găsire a unui traseu pe o suprafață împărțită în pătrate și pentru care există restricții de deplasare de la un pătrat la altul.



1. **Problema bilei**. Un teren este împărțit în mai multe zone, fiecare zonă având o înălțime h . Pentru simplificare, vom considera fiecare zonă ca fiind un pătrat cu aceeași dimensiune. Terenul poate fi reprezentat sub forma unei matrice cu n linii și m coloane, fiecare element al matricei reprezentând înălțimea unei zone de teren. O bilă se găsește într-o zonă cu coordonatele x (numărul liniei) și y (numărul coloanei). Ea se poate rostogoli numai către o zonă care are înălțimea mai mică decât cea a zonei în care se găsește. Să se genereze toate posibilitățile de rostogolire a bilei – până la marginea terenului. **Indicație.** Construirea soluției traseului are următoarele caracteristici:

- O soluție este formată din coordonatele zonelor prin care se rostogolește bila.
- Fiecare soluție are un număr variabil de elemente. Primul element al soluției conține coordonatele zonei de plecare a bilei, iar ultimul element al soluției conține coordonatele unei zone de la marginea terenului.
- Bila se poate rostogoli în 8 direcții (corespunzătoare celor 8 pătrate adiacente păratului în care se găsește), fiecare direcție de deplasare corespunzând unei constante de deplasare care se adaugă la coordonata x , respectiv y , a păratului în care se găsește bila.
- **Condiția de continuare** a construirii soluției este ca păratul în care a ajuns bila să aibă înălțimea mai mică decât a păratului anterior.

2. **Problema capcanelor**. Un teren este împărțit în mai multe zone. Pentru simplificare vom considera fiecare zonă ca fiind un pătrat cu aceeași dimensiune. Terenul poate fi reprezentat sub forma unei matrice cu n linii și m coloane, fiecare element al matricei reprezentând un pătrat de teren. Anumite pătrate conțin diverse capcane ascunse. O persoană se găsește în păratul cu coordonatele x_1 (numărul liniei) și y_1 (numărul coloanei) și trebuie să ajungă în păratul cu coordonatele x_2 și y_2 deplasându-se ortogonal și fără să calce în pătratele cu capcane. Datele se citesc dintr-un fișier text, astfel: de pe primul rând, n , m și coordonatele păratului de pornire și ale păratului destinație, iar de pe următoarele rânduri – perechile de coordonate ale păratelor cu capcane. Să se găsească traseele pe care trebuie să le urmeze persoana respectivă. **Indicație.** Construirea soluției traseului are următoarele caracteristici:

- O soluție este formată din coordonatele zonelor prin care trece persoana.
- Fiecare soluție are un număr variabil de elemente. Primul element al soluției conține coordonatele păratului din care pleacă persoana, iar ultimul element al soluției conține coordonatele păratului în care trebuie să ajungă.
- Persoana se poate deplasa în 4 direcții (corespunzătoare celor 4 pătrate aflate pe diagonala păratului în care se găsește), fiecare direcție de deplasare corespunzând unei constante de deplasare care se adaugă la coordonata x , respectiv y , a păratului în care se găsește persoana.
- **Condiția de continuare** a construirii soluției este ca păratul în care a ajuns persoana să nu conțină capcane.

1.4. Metoda „Divide et Impera“

1.4.1. Descrierea metodei „Divide et Impera“

Metoda divide et impera se poate folosi pentru problemele care **pot fi descompuse în subprobleme similare cu problema inițială** (care se rezolvă prin aceeași metodă) și care prelucrează multimi de date de dimensiuni mai mici, **independente unele de altele** (care folosesc multimi de date de intrare disjuncte).

Studiu de caz

Scop: identificarea problemelor care pot fi descompuse în subprobleme similare care folosesc multimi de date de intrare disjuncte.

Enunțul problemei 1: Să se calculeze suma elementelor dintr-un vector v care conține numere întregi.

Mulțimea datelor de intrare o reprezintă cele n elemente ale vectorului v . Ele pot fi divizate în câte două submultimi disjuncte, prin divizarea mulțimii indicilor în două submultimi. Mulțimea inițială a indicilor este determinată de primul indice (s) și de ultimul indice (d), iar intervalul indicilor care se divizează este $[s, d]$. El se divizează în două submultimi disjuncte, $[s, m]$ și $[m+1, d]$, unde m este indicele din mijlocul intervalului: $m=(s+d)/2$. Astfel, problema inițială este descompusă în două subprobleme, fiecare dintre ele constând în calcularea sumei numerelor dintr-o submulțime de elemente (care corespunde unui subinterval al indicilor). Descompunerea continuă până când fiecare submulțime conține un singur element – și se poate calcula suma, obținându-se soluția subproblemei.

Enunțul problemei 2: Să se calculeze suma $1 \times 2 + 2 \times 3 + \dots + n \times (n+1)$.

Mulțimea datelor de intrare o reprezintă primele n numere naturale. Mulțimea inițială este determinată de primul număr ($s=1$) și de ultimul număr ($d=n$), iar intervalul care se divizează este $[s, d]$. El se divizează în două submultimi disjuncte, $[s, m]$ și $[m+1, d]$, unde m este numărul din mijlocul intervalului: $m=(s+d)/2$. Astfel, problema inițială este descompusă în două subprobleme, fiecare dintre ele constând în calcularea sumei produselor dintre două numere consecutive dintr-o submulțime de elemente (care corespunde unui subinterval al numerelor). Descompunerea continuă până când fiecare submulțime conține un singur element – și se poate calcula produsul, care se va adăuga la sumă, pentru a obține soluția subproblemei.

Enunțul problemei 3: Să se genereze termenul n al sirului lui Fibonacci.

Sirul lui Fibonacci este definit recursiv: $f_1=1$, $f_2=1$ și $f_n = f_{n-2} + f_{n-1}$, pentru $n \geq 3$. Problema determinării termenului n al sirului lui Fibonacci se poate descompune în două subprobleme: determinarea termenului $n-1$ și determinarea termenului $n-2$. Descompunerea continuă până când trebuie determinați termenii f_1 și f_2 , a căror valoare este cunoscută.



Metoda **Divide et impera** se bazează pe descompunerea unei probleme în subprobleme similare, prin intermediul unui proces recursiv. Procesul recursiv de descompunere a unei subprobleme în alte subprobleme continuă până se obține o subproblemă cu rezolvarea imediată (cazul de bază), după care se compun soluțiile subproblemelor până se obține soluția problemei initiale.

Pașii algoritmului sunt:

- PAS1.** Se descompune problema în subprobleme similare problemei initiale, de dimensiuni mai mici, independente unele de altele (care folosesc multimi de date de intrare disjuncte – d_i).
- PAS2.** Dacă subproblema permite rezolvarea imediată (corespunde cazului de bază), atunci se rezolvă obținându-se soluția s ; altfel, se revine la Pas1.
- PAS3.** Se combină soluțiile subproblemelor în care a fost descompusă (s_i) o subproblemă, până când se obține soluția problemei initiale.

1.4.2. Implementarea metodei Divide et Impera

Deoarece subproblemele în care se descompune problema sunt similare cu problema inițială, algoritmul **divide et impera** poate fi implementat **recursiv**. Subprogramul recursiv **divide_et_impera(d,s)**, unde d reprezintă dimensiunea subproblemei (corespunde mulțimii datelor de intrare), iar s soluția subproblemei, poate fi descris în pseudocod astfel:

```

divide_et_impera(d,s)
început
    dacă dimensiunea d corespunde unui caz de bază
        atunci se determină soluția s a problemei;
    altfel
        pentru i=1,n execută
            se determină dimensiunea d_i a subproblemei P_i;
            se determină soluția s_i a subproblemei P_i prin
                apelul divide_et_impera(d_i,s_i);
        sfărșit_pentru;
        se combină soluțiile s_1, s_2, s_3, ..., s_n;
    sfărșit_dacă;
sfărșit;

```

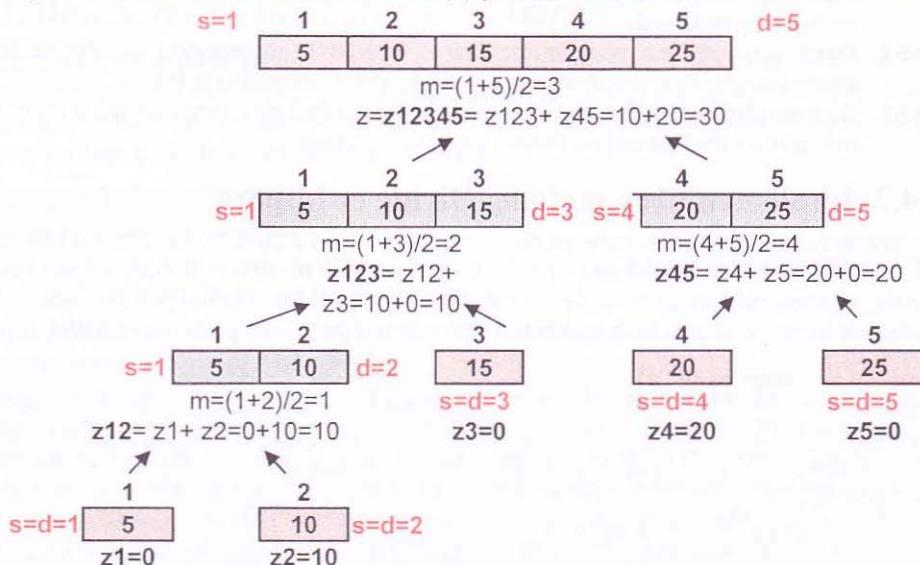
Implementarea acestui algoritm în limbajul C++ se face astfel:

```

/*declarații globale pentru datele de intrare ce vor fi divizate în sub-
multimi disjuncte pentru subproblemele în care se descompune problema*/
void divizeaza(<parametri: submultimile>)
{//se divizează mulțimea de date de intrare în submultimi disjuncte d_i}
void combina(<parametri: soluțiile s_i care se combină>)
{//se combină soluțiile obținute s_i}
void dei(<parametri: mulțimea de date d și soluția s>)
{//declarații de variabile locale
if (<este caz de bază>) //se obține soluția corespunzătoare subproblemei}
else
    {divizeaza(<parametri: k submultimi>);
    for (i=1;i=k;i++)
        dei(<parametri: mulțimea de date d_i și soluția s_i>);
    combina(<parametri: soluțiile s_i>);}
void main()
{//declarații de variabile locale
//se citesc datele de intrare ale problemei - mulțimea d
dei(<parametri: mulțimea de date d și soluția s>);
//se afișează soluția problemei - s}

```

Exemplul 1. Să se calculeze **suma elementelor pare** dintr-un vector **v** care conține numere întregi. Numărul de elemente ale vectorului (**n**) și elementele lui se citesc de la tastatură.



Implementarea metodei **divide et impera** în acest exemplu se face astfel:

- Subprogramul **divizeaza()** – Numărul de subprobleme în care se descompune problema este 2 (**k=2**). Multimea datelor de intrare este divizată în două submultimi disjuncte, prin divizarea multimii indicilor în două submultimi disjuncte de indici, adică multimea indicilor **[s,d]** (unde **s** este primul indice, iar **d** ultimul indice) este divizată în două submultimi disjuncte **[s,m]** și **[m+1,d]**, unde **m** este indicele din mijlocul intervalului: $m = (s+d)/2$. În subprogram, procesul de divizare constă în determinarea mijlocului intervalului – **m**.
- Subprogramul **combina()** – Combinarea soluției înseamnă adunarea celor două sume obținute prin rezolvarea celor două subprobleme. În subprogram sunt combinate cele două valori obținute din prelucrarea celor două intervale, adică se adună cele două valori **x** și **y**, obținându-se soluția **z**.
- Subprogramul **dei()** – O subproblemă corespunde cazului de bază atunci când submultimea conține un singur element (se poate calcula suma, obținându-se soluția subproblemiei). Dacă s-a terminat procesul recursiv (prin procesul de divizare, cele două capete ale intervalului au ajuns să fie identice), atunci se prelucrează cazul de bază (se calculează suma în variabila **z**, corespunzătoare soluției, astfel: dacă numărul **v[s]** este par, atunci suma va fi chiar numărul; altfel, are valoarea 0); altfel, se apelează subprogramul pentru divizarea intervalului, se apelează subprogramul **dei()** pentru primul interval, se apelează subprogramul **dei()** pentru al doilea interval și se combină cele două rezultate.

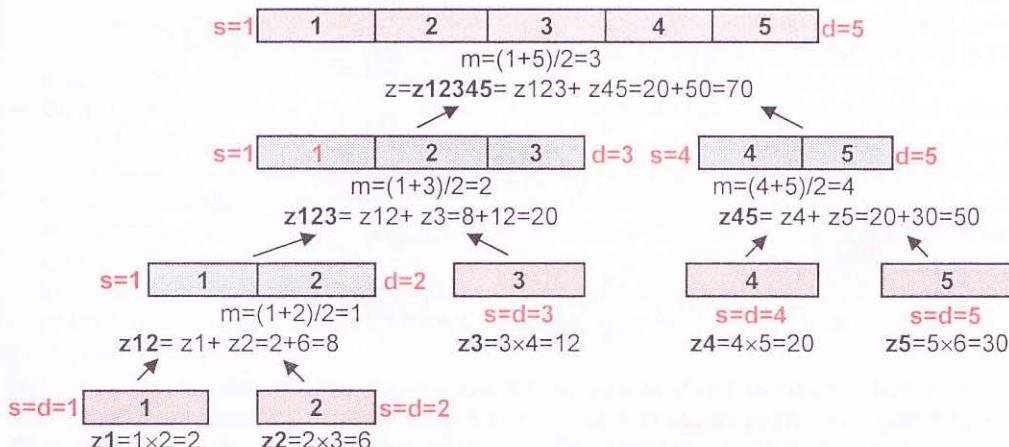
```
#include<iostream.h>
int v[100], n;
void divizeaza(int s, int d, int &m) {m=(s+d)/2;}
void combina(int x, int y, int &z) {z=x+y;}
void dei(int s, int d, int &z)
{int m,x1,x2;
 if (d==s)
```

```

if (v[s]%2==0) z=v[s]; else z=0;
else
    {divizeaza(s,d,m); dei(s,m,x1); dei(m+1,d,x2); combina(x1,x2,z);}
void main()
{int i,z; cout<<"n= ";cin>>n;
for(i=1;i<=n;i++) {cout<<"v["<<i<<"]="; cin>>v[i];}
dei(1,n,z); cout<<"suma=" <<z;}

```

Exemplul 2: Să se calculeze suma $1 \times 2 + 2 \times 3 + \dots + n \times (n+1)$.



Implementarea metodei **divide et impera** în acest exemplu se face astfel:

- Subprogramul **divizeaza()** – Numărul de subprobleme în care se descompune problema este 2 ($k=2$). Multimea datelor de intrare este divizată în două submultimi disjuncte, prin divizarea multimei primelor n numere naturale în două submultimi disjuncte, adică multimea $[s,d]$ (unde s este primul număr din multime, iar d ultimul număr din multime) este divizată în două submultimi disjuncte, $[s,m]$ și $[m+1,d]$, unde m este numărul din mijlocul intervalului: $m=(s+d)/2$. În subprogram, procesul de divizare constă în determinarea mijlocului intervalului, m .
- Subprogramul **combina()** – Combinarea soluției înseamnă adunarea celor două sume obținute prin rezolvarea celor două subprobleme. În subprogram sunt combinate cele două valori obținute din cele două intervale (se adună cele două valori, x și y) obținându-se soluția z .
- Subprogramul **dei()** – O subproblemă corespunde **cazului de bază** atunci când submultimea conține un singur element (se poate calcula termenul sumei – produsul celor două numere consecutive – obținându-se soluția subproblemei). Dacă s-a terminat procesul recursiv (prin procesul de divizare, cele două capete ale intervalului au ajuns să fie identice), **atunci** se prelucrează cazul de bază (se calculează produsul în variabila z , corespunzătoare soluției); **altfel**, se apelează subprogramul pentru divizarea intervalului, se apelează subprogramul **dei()** pentru primul interval, se apelează subprogramul **dei()** pentru al doilea interval și se combină cele două rezultate.

```

#include<iostream.h>
int n;
void divizeaza(int s,int d,int &m) {m=(s+d)/2;}
void combina(int x,int y,int &z) {z=x+y;}

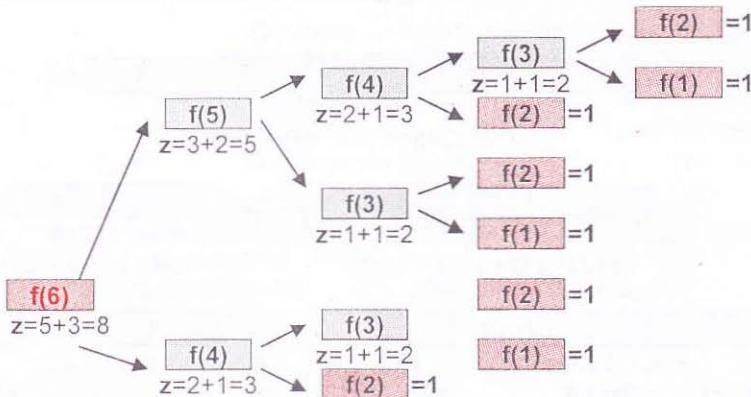
```

```

void dei(int s,int d,int &z)
{int m,x1,x2;
 if (d==s) z=s*(s+1);
 else {divizeaza(s,d,m); dei(s,m,x1); dei(m+1,d,x2); combina(x1,x2,z);}
void main()
{int z; cout<<"n= ";cin>>n; dei(1,n,z); cout<<"suma ="<<z; }

```

Exemplul 3: Să se calculeze termenul n al **șirului lui Fibonacci**.



Implementarea metodei divide et impera în acest exemplu se face astfel:

- Subprogramul **divizeaza()** – Numărul de subprobleme în care se descompune problema este 2 ($k=2$): determinarea termenului $n-1$ și determinarea termenului $n-2$. Descompunerea se face implicit, prin parametrul cu care se apelează subprogramul **dei()**, și subprogramul de divizare nu mai este necesar.
- Subprogramul **combina()** – Combinarea soluției înseamnă adunarea celor doi termeni ai șirului (x_1 și x_2), obținându-se soluția z .
- Subprogramul **dei()** – O subproblemă corespunde **cazului de bază** atunci când s-a ajuns la un termen direct calculabil. Dacă s-a terminat procesul recursiv (prin procesul de divizare s-a ajuns la termenul f_1 sau f_2), atunci se prelucrează cazul de bază; **altfel**, se apelează subprogramul **dei()** pentru primul termen al descompunerii, se apelează subprogramul **dei()** pentru al doilea termen al descompunerii și se combină cele două rezultate.

```

#include<iostream.h>
int n;
void combina(int x1,int x2,int &z) {z=x1+x2;}
void dei(int n,int &z)
{int x1,x2;
 if (n==1 || n==2) z=1;
 else {dei(n-1,x1); dei(n-2,x2); combina(x1,x2,z);}}
void main()
{int z; cout<<"n= "; cin>>n; dei(n,z); cout<<z; }

```

Atenție

În acest exemplu, problema a fost descompusă în probleme care nu sunt independente unele de altele și, în apelurile recursive, aceeași subproblemă este rezolvată de mai multe ori: $f(4)$, $f(3)$, $f(2)$, $f(1)$. Pentru acest tip de problemă **nu se recomandă** metoda **divide et impera** deoarece nu este eficientă.

Exemplul 4: Să se determine simultan valoarea **minimă** și valoarea **maximă** dintr-un vector v care conține numere întregi. Numărul de elemente ale vectorului (n) și elementele lui se citesc de la tastatură.

Implementarea metodei **divide et impera** în acest exemplu se face astfel:

- Subprogramul **divizeaza()** – Numărul de subprobleme în care se descompune problema este 2 ($k=2$). Divizarea mulțimii datelor de intrare se face la fel ca la exemplul pentru calcularea sumei elementelor unui vector și subprogramele sunt identice.
- Subprogramul **combina()** – Combinarea soluției înseamnă determinarea minimului (z_1) și a maximului (z_2) dintre cele două valori minime (x_1 și y_1), respectiv maxime (x_2 și y_2), obținute prin rezolvarea celor două subprobleme. În subprogram sunt combinate cele două perechi de valori obținute din cele două intervale. Dacă $x_1 > y_1$, atunci minimul (z_1) este y_1 ; altfel, este x_1 . Dacă $x_2 > y_2$, atunci maximul (z_2) este x_2 ; altfel, este y_2 .
- Subprogramul **dei()** – O subproblemă corespunde **cazului de bază** atunci când submulțimea conține un singur element (se pot calcula minimul și maximul; atât minimul cât și maximul vor avea valoarea elementului). Dacă s-a terminat procesul recursiv (prin procesul de divizare cele două capete ale intervalului au ajuns să fie identice), atunci se prelucrează cazul de bază (se calculează minimul și maximul în variabilele z_1 și z_2 corespunzătoare soluției); altfel, se apelează subprogramul pentru divizarea intervalului, se apelează subprogramul **dei()** pentru primul interval, se apelează subprogramul **dei()** pentru al doilea interval și se combină cele două rezultate.

```
#include<iostream.h>
int v[100], n;
void divizeaza(int s, int d, int &m) {m=(s+d)/2;}
void combina(int x1, int y1, int &z1, int x2, int y2, int &z2)
{if (x1>y1) z1=y1; else z1=x1;
 if (x2>y2) z2=x2; else z2=y2;}
void dei(int s, int d, int &z1, int &z2) //z1 - minim, z2 - maxim
{int m, x1, x2, y1, y2;
 if (d==s) z1=z2=v[s];
 else {divizeaza(s, d, m);
       dei(s, m, x1, x2); //x1 - minim, x2 - maxim
       dei(m+1, d, y1, y2); //y1 - minim, y2 - maxim
       combina(x1, y1, z1, x2, y2, z2);}}
void main()
{int i, z1, z2; cout<<"n= "; cin>>n;
 for(i=1; i<=n; i++) {cout<<"v["<<i<<"]="; cin>>v[i];}
 dei(1, n, z1, z2);
 cout<<"minimul ="<<z1<<endl<<"maximul ="<<z2<<endl;}
```

Exemplul 5: Să se calculeze **suma a două polinoame**. Gradele celor două polinoame, n și m, și coeficienții celor două polinoame se citesc de la tastatură. Coeficienții celor două polinoame se memorează în vectorii p și q, iar coeficienții polinomului sumă se memorează în vectorul r.

Implementarea metodei **divide et impera** în acest exemplu se face astfel:

- Subprogramul **divizeaza()** – Numărul de subprobleme în care se descompune problema este 2 ($k=2$). Deoarece există două mulțimi de date de intrare (vectorii p și q care memorează coeficienții celor două polinoame), se va lua ca reper mulțimea cu cele mai multe elemente. Mulțimea datelor de intrare este divizată în două submulțimi disjuncte,

prin divizarea multimii indicilor în două submultimi disjuncte de indici, adică multimea indicilor $[s, d]$ (unde s este primul indice, iar d este ultimul indice – $d = \text{maxim}(n, m) + 1$) este divizată în două submultimi disjuncte $[s, mijl]$ și $[mijl + 1, d]$, unde $mijl$ este indicele din mijlocul intervalului: $mijl = (s+d)/2$. Procesul de divizare este identic cu cel de la exemplele anterioare.

- Subprogramul **combina()** – Deoarece în cazul de bază se determină unul dintre coeficientii polinomului sumă care se scrie direct în vectorul r , acest subprogram nu mai este necesar.
- Subprogramul **dei()** – O subproblemă corespunde **cazului de bază** atunci când submulțimea conține un singur element (se poate calcula coeficientul polinomului sumă). Dacă s-a terminat procesul recursiv (prin procesul de divizare, cele două capete ale intervalului au ajuns să fie identice), **atunci** se prelucrează cazul de bază, prin care se calculează coeficientul polinomului sumă pentru acel indice (dacă termenul care se calculează are indicele mai mic decât gradul minim al polinoamelor, **atunci** el este egal cu suma coeficientilor celor două polinoame; **altfel**, dacă polinomul p are gradul mai mic decât al polinomului q , **atunci** el este egal cu coeficientul polinomului q ; **altfel**, el este egal cu coeficientul polinomului p); **altfel**, se apelează subprogramul pentru divizarea intervalului, se apelează subprogramul **dei()** pentru primul interval și apoi pentru al doilea interval.

```
#include<iostream.h>
int p[10], q[10], r[10], n, m;
int maxim(int x, int y) {if (x > y) return x; else return y;}
int minim(int x, int y) {if (x > y) return y; else return x;}
void divizeaza(int s, int d, int &mijl) {mijl = (s+d)/2;}
void dei(int s, int d)
{int mijl;
 if (d==s)
    if (d<=minim(n, m)) r[d]=p[d]+q[d];
    else if(n<m) r[d]=q[d];
    else r[d]=p[d];
 else {divizeaza(s, d, mijl); dei(s, mijl); dei(mijl+1, d);}
}
void main()
{int i; cout<<"n= "; cin>>n; cout<<"m= "; cin>>m;
 for(i=1; i<=n+1; i++) {cout<<"p("<<i-1<<")= "; cin>>p[i];
 for(i=1; i<=m+1; i++) {cout<<"q("<<i-1<<")= "; cin>>q[i];
 dei(1, maxim(n, m)+1);
 for(i=maxim(n, m)+1; i>=1; i--)
    if (r[i]!=0)
       {if (r[i]==1) {cout<<r[i]; if (i!=1) cout<<"*";}
        if (i>2) cout<<"x^"<<i-1; else if (i==2) cout<<"x";
        if (i!=1) cout<<" + ";}}}
```



Temă

1. Programul următor afișează, în ordine inversă, elementele unui vector. Explicați cum a fost folosită metoda **divide et impera** pentru a rezolva problema.

```
#include<iostream.h>
int v[100], n;
void divizeaza(int s, int d, int &m)
{m=(s+d)/2;}
```

```

void dei(int s,int d)
{int m;
 if (d==s) cout<<v[s]<<" ";
 else {divizeaza(s,d,m); dei(m+1,d); dei(s,m);}
void main()
{int i; cout<<"n= ";cin>>n;
 for(i=1;i<=n;i++) {cout<<"a("<<i<<")= "; cin>>v[i];}
 dei(1,n);}

```

2. Determinați ce calculează programul următor. Explicați cum a fost folosită metoda **divide et impera** pentru a rezolva problema.

```

#include<iostream.h>
int n;
void divizeaza(int s,int d,int &m) {m=(s+d)/2;}
void combina(int x,int y,int &z) {z=x+y;}
void dei(int s,int d,int &z)
{int m,x1,x2;
 if (d==s) {if(s%2==0) z=-s*5;else z=s*5;}
 else
 {divizeaza(s,d,m); dei(s,m,x1); dei(m+1,d,x2); combina(x1,x2,z);}
void main()
{int z; dei(1,20,z); cout<<"suma = "<<z;}

```



Tema

Scripteți următoarele programe, în care folosiți metoda **divide et impera**.

Valorile pentru datele de intrare se citesc de la tastatură.

1. Să se calculeze $n!$.
2. Să se calculeze simultan produsul și suma a n numere memorate într-un vector.
3. Să se calculeze suma $1+1 \times 2 + 1 \times 2 \times 3 + \dots + 1 \times 2 \times 3 \times \dots \times n$.
4. Să se numere elementele pare dintr-un vector.
5. Să se verifice dacă un vector conține numai numere pozitive sau numai numere negative.
6. Să se calculeze c.m.m.d.c. a n numere memorate într-un vector.
7. Să se determine numărul de aparții ale unei valori x într-un vector.
8. Să se calculeze valoarea unui polinom $P(x)$ într-un punct x precizat.
9. Într-o matrice cu n linii și m coloane, să se interschimbe coloana p cu coloana q .
10. Într-o matrice pătrată cu dimensiunea n să se interschimbe linia p cu coloana q .
11. Într-o matrice pătrată cu dimensiunea n să se interschimbe diagonala principală cu diagonală secundară.
12. Să se determine simultan valoarea minimă și valoarea maximă, dintr-o matrice cu n linii și m coloane.

Complexitatea algoritmului **divide et impera**

Metoda **divide et impera** se bazează pe rezolvarea recursivă a subproblemelor în care este divizată problema inițială.

Atunci când un algoritm conține un apel recursiv, timpul său de execuție este dat de o formulă recursivă care calculează timpul de execuție al algoritmului pentru o dimensiune n a datelor de intrare, cu ajutorul timpilor de execuție pentru dimensiuni mai mici. Timpul de execuție al unui algoritm care folosește metoda **divide et impera** se bazează pe calculul timpilor de execuție ai celor trei etape de rezolvare a problemei. Dacă:

- problema inițială se divizează în a subprobleme, pentru care dimensiunea datelor de intrare reprezintă **1/b** din dimensiunea problemei inițiale;
 - timpul de execuție a problemei inițiale este **T(n)**;
 - timpul necesar pentru a divide problema în subprobleme este **D(n)**;
 - timpul necesar pentru combinarea soluțiilor subproblemelor, pentru a obține soluția problemei, este **C(n)**;
 - timpul necesar pentru rezolvarea cazului de bază este **Θ(1)**;
- atunci se obține funcția pentru timpul de execuție care este prezentată alăturat.

$$T(n) = \begin{cases} \Theta(1) & \text{pentru cazul de bază} \\ axT(n/b) + D(n) + C(n) & \text{în caz contrar} \end{cases}$$

De exemplu, pentru a calcula suma elementelor unui vector cu **n** elemente, problema se descompune în două subprobleme (**a=2**) și dimensiunea datelor de intrare pentru o subproblemă reprezintă jumătate din dimensiunea datelor inițiale (**b=2**). Pentru divizarea problemei în subprobleme, se calculează mijlocul intervalului de indici și **O(D(n))=Θ(1)**. Pentru combinarea celor două soluții ale fiecărei subprobleme, se adună cele două valori, și **O(C(n))=Θ(1)**. Considerăm că **n=2^k**. Rezultă că:

$$\begin{aligned} T(n) &= T(2^k) + 2 \times \Theta(1) = 2 \times (T(2^{k-1}) + 2 \times \Theta(1)) + 2 \times \Theta(1) = 2 \times (T(2^{k-1}) + 2^2 \times \Theta(1)) = \\ &= 2 \times (2 \times T(2^{k-2}) + 2 \times \Theta(1)) + 2^2 \times \Theta(1) = 2^2 \times T(2^{k-2}) + 2^3 \times \Theta(1) = \dots = \\ &= 2^k \times T(2^0) + 2^{k+1} \times \Theta(1) = 2^k \times \Theta(1) + 2^{k+1} \times \Theta(1) = 2^k \times 3 = n \times 3. \end{aligned}$$

Ordinul de complexitate al algoritmului este **O(n)**. Algoritmul iterativ pentru rezolvarea acestei probleme are ordinul de complexitate **O(n)**.

Observație. Pentru acest gen de probleme, metoda iterativă este mai eficientă. Rezolvarea acestui tip de probleme cu ajutorul metodei divide et impera a avut numai un rol scolaristic, pentru înțelegerea metodei și a implementării ei.

Metoda divide et impera **se recomandă** în următoarele cazuri:

- algoritmul obținut este **mai eficient** decât algoritmul clasic (iterativ) – de exemplu, algoritmul de căutare într-un vector sortat și algoritmii pentru sortarea unui vector;
- rezolvarea problemei prin divizarea ei în subprobleme este **mai simplă** decât rezolvarea clasică (iterativă) – de exemplu, problema turnurilor din Hanoi și generarea unor modele fractale.

1.4.3. Căutarea binară

Să se caute, într-un sir de numere întregi ordonate strict crescător (sau varianta strict descrescător), poziția în care se găsește în sir o valoare **x** citită de la tastatură. Dacă valoarea nu se găsește în sir, să se afișeze un mesaj de informare.

Algoritmul de căutare secvențială într-un vector are ordinul de complexitate **O(n)**. Pornind de la faptul că vectorul este un vector particular (este ordonat strict crescător sau strict descrescător), se poate folosi metoda **divide et impera**. Pentru un vector ordonat strict crescător, **pașii algoritmului sunt**:

- PAS1.** Se divizează vectorul **v** în doi subvectori, prin divizia multimi indicilor **[s,d]** (unde **s** este indicele primului element, iar **d** indicele ultimului element) în două submultimi disjuncte, **[s,m]** și **[m+1,d]**, unde **m** este indicele din mijlocul intervalului: **m=(s+d)/2**.
- PAS2.** Dacă elementul situat pe poziția din mijloc (**v[m]**) este valoarea **x**, atunci problema este rezolvată și poziția este **m**; altfel, dacă elementul din mijlocul vectorului este mai mic decât valoarea **x**, atunci căutarea se face printre elementele cu indicii în mulțimea **[s,m]**; altfel, căutarea se face printre elementele cu indicii din

mulțimea $[m+1, d]$. Pasul 2 se execută până când se găsește elementul sau până când vectorul nu mai poate fi împărțit în subvectori.

```
#include<iostream.h>
int v[100], n, x;
void divizeaza(int s, int d, int &m) {m=(s+d)/2;}
void cauta(int s, int d, int &z)
{int m;
 if (d>s) {divizeaza(s, d, m);
    if (v[m]==x) z=m;
    else if (x>v[m]) cauta(m+1, d, z);
    else cauta(s, m, z);}}
void main()
{int i, z=0; cout<<"n= "; cin>>n; cout<<"x= "; cin>>x;
 for(i=1; i<=n; i++) {cout<<"v["<<i<<"]="; cin>>v[i];}
 cauta(1, n, z);
 if(z==0) cout<<"nu există elementul "<<x<<" în vector";
 else cout<<"există pe poziția "<<z;}
```

Complexitatea algoritmului de căutare binară. Pentru divizarea problemei în subprobleme, se calculează mijlocul intervalului de indici și $O(D(n))=\Theta(1)$. Deoarece căutarea se face numai într-unul dintre cei doi subvectori (problema inițială se rezolvă prin rezolvarea uneia dintre cele două subprobleme) și $a=1$, formula recurrentă a timpului de execuție este $T(n)=T(n/2)+\Theta(1)$. Considerăm că $n=2^k$ ($k=\log_2 n$). Rezultă că:

$$\begin{aligned} T(n) &= T(2^k) + \Theta(1) = (T(2^{k-1}) + \Theta(1)) + \Theta(1) = T(2^{k-1}) + 2 \times \Theta(1) = \dots = \\ T(2^0) &+ (k+1) \times \Theta(1) = \Theta(1) + (k+1) \times \Theta(1) = (k+2) \times \Theta(1) = \log_2 n \times \Theta(1). \end{aligned}$$

Ordinul de complexitate al algoritmului este **$O(\log_2 n)$** .

Aplicarea algoritmului de căutare binară

Exemplu. Să se determine, cu o precizie de 4 zecimale, rădăcina reală, din intervalul $[0, 1]$, a ecuației $x^3+x-1=0$.

S-a identificat pentru această ecuație o rădăcină în intervalul $[0, 1]$ și ne propunem să localizăm această rădăcină, în limitele unei precizii de 4 zecimale, printr-o valoare x . Pentru căutarea valorii x se va folosi **metoda bisectiei** – care constă în reducerea intervalului de căutare prin înjumătățirea repetată și selectarea subintervalului în care se găsește rădăcina. Intervalul $[s, d]$ este împărțit în două subintervale, $[s, m]$ și $[m, d]$, unde $m=(s+d)/2$. Căutarea rădăcinii se va face în subintervalul în care funcția $f(x)=x^3+x-1$ își schimbă semnul, astfel: dacă $f(s)*f(m)<0$, atunci căutarea continuă în intervalul $[s, m]$; altfel, căutarea continuă în subintervalul $[m, d]$. Procesul recursiv este întrerupt când se ajunge la intervalul $[s, d]$ pentru care $d-s< r$, unde r este eroarea acceptată pentru o precizie de 4 zecimale și are valoarea 0,0001.

```
#include<iostream.h>
#include<iomanip.h>
const float r=0.0001;
float f(float x) {return x*x*x+x-1;}
void divizeaza(float s, float d, float &m) {m=(s+d)/2;}
void radacina(float s, float d, float &z)
{float m;
 if (d-s<r) z=(s+d)/2;
 else
```

```

{divizeaza(s,d,m);
if (f(s)*f(m)<0) radacina(s,m,z);
else radacina(m,d,z);}

void main()
{float z=0; radacina(0,1,z); cout<<"radacina= "<<z<<endl;
cout<<"f(x)= "<<setiosflags(ios::fixed)<<f(z); }

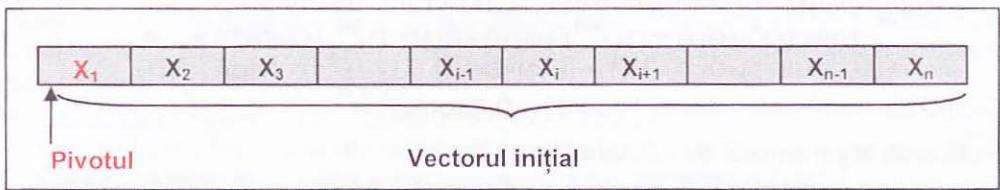
```

Temă

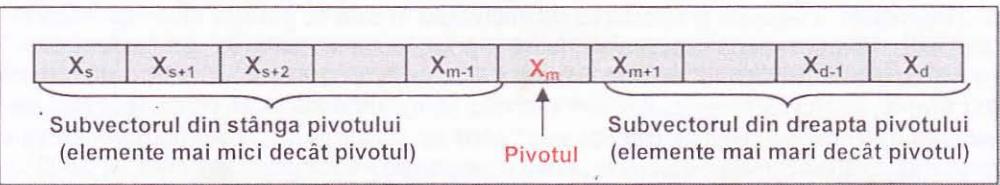
1. Să se caute, într-un sir de numere întregi, în care mai întâi se găsesc numerele pare și apoi numerele impare, poziția în care se găsește, în sir, o valoare x citită de la tastatură.
2. Să se calculeze radicalul de ordinul 2 din numărul a , cu o aproximatie de 4 zecimale, fără a folosi funcția matematică de sistem `sqrt()`. Să se compare rezultatul obținut cu rezultatul furnizat de funcția matematică de sistem. **Indicație.** Pornind de la ecuația $x^2-a=0$, se va identifica intervalul în care se găsește soluția și apoi se va localiza, cu o precizie de 4 zecimale, rădăcina reală din acest interval.
3. Să se calculeze partea întreagă a radicalului de ordinul 3 din numărul a fără a folosi funcția matematică de sistem.
4. Să se calculeze radicalul de ordinul 3 din numărul a , cu o aproximatie de 4 zecimale.

1.4.4. Sortarea rapidă (QuickSort)

Prin această metodă de sortare se execută următoarele operații prin care sunt rearanjate elementele din cadrul vectorului:



- Primul element din vector, numit **pivot**, este mutat în cadrul vectorului pe poziția pe care trebuie să se găsească în vectorul sortat.
- Toate elementele mai mici decât el vor fi mutate în vector în fața sa.
- Toate elementele mai mari decât el vor fi mutate în vector după el.



De exemplu, dacă vectorul conține elementele $\{3, 4, 1, 5, 2\}$, după executarea operațiilor precizate vectorul va fi $\{2, 1, 3, 5, 4\}$.

Folosind metoda **divide et impera** problema inițială va fi descompusă în subprobleme, astfel:

- PAS1.** Se rearanjează vectorul, determinându-se poziția în care va fi mutat pivotul (m).
- PAS2.** Problema inițială (sortarea vectorului inițial) se descompune în două subprobleme, prin descompunerea vectorului în doi subvectori: vectorul din stânga pivotului și vectorul din dreapta pivotului, care vor fi sortați prin aceeași metodă. Acești subvectori, la rândul lor, vor fi și ei rearanjați și împărțiți de pivot în doi subvectori etc.

PAS3. Procesul de descompunere în subprobleme va continua până când, prin descompunerea vectorului în subvectori, se vor obține vectori care conțin un singur element.

Subprogramele specifice algoritmului **divide et impera** vor avea următoarea semnificație:

- În subprogramul **divizeaza()** se va rearanja vectorul și se va determina poziția pivotului x_m , care va fi folosită pentru divizarea vectorului în doi subvectori: $[x_s, x_{m-1}]$ și $[x_{m+1}, x_d]$.
- Subprogramul **combina()** nu mai este necesar, deoarece combinarea soluțiilor se face prin rearanjarea elementelor în vector.

În subprogramul **divizeaza()** vectorul se parcurge de la ambele capete către poziția în care trebuie mutat pivotul. Se vor folosi doi indici: i – pentru parcurgerea vectorului de la începutul lui către poziția pivotului (i se va incrementa) și j – pentru parcurgerea vectorului de la sfârșitul lui către poziția pivotului (j se va decrementa). Cei doi indici vor fi inițializați cu capetele vectorului ($i=s$, respectiv $j=d$) și se vor deplasa până când se întâlnesc, adică atât timp cât $i < j$. În momentul în care cei doi indici s-au întâlnit înseamnă că operațiile de rearanjare a vectorului s-au terminat și pivotul a fost adus în poziția corespunzătoare lui în vectorul sortat. Această poziție este i (sau j) și va fi poziția m de divizare a vectorului.

În exemplele următoare sunt prezentate două versiuni pentru subprogramul **divizeaza()**:

Versiunea 1. Se folosesc variabilele logice: pi , pentru parcurgerea cu indicele i , și pj , pentru parcurgerea cu indicele j . Ele au valoarea: 1 – se parurge vectorul cu acel indice, și 0 – nu se parurge vectorul cu acel indice; cele două valori sunt complementare.

```
#include<iostream.h>
int x[100],n;
void schimb(int &a, int &b){int aux=a; a=b; b=aux;}
void divizeaza(int s,int d,int &m)
{int i=s,j=d,pi=0,pj=1;
// pivotul fiind pe pozitia s, parcurgerea incepe cu indicele j
while (i<j)
{if (x[i]>x[j]) {schimb(x[i],x[j]); schimb(pi,pj);}
i=i+pi; j=j-pj;}
m=i;}
void QuickSort(int s,int d)
{int m;
if (s<d) {divizeaza(s,d,m);
QuickSort(s,m-1);
QuickSort(m+1,d);}}
void main()
{int i; cout<<"n= ";cin>>n;
for(i=1;i<=n;i++) {cout<<"x["<<i<<"]= ";cin>>x[i];}
QuickSort(1,n);
cout<<"vectorul sortat"<<endl; for(i=1;i<=n;i++) cout<<x[i]<< " ";
```

Cei doi indici i și j sunt inițializați cu extremitățile vectorului ($i=1$; $j=5$) și parcurgerea începe cu indicele j ($pi=0$; $pj=1$)

	1	2	3	4	5

	1	2	3	4	5

Se compară pivotul (3) cu ultimul element (2). Deoarece pivotul este mai mic, cele două valori se interschimbă, și se schimbă și modul de parcurgere ($pi=1$; $pj=0$ – avanseză indicele i).

Se compară elementul din poziția i (4) cu elementul din poziția j (3). Deoarece 4 este mai mare decât 3, cele două valori se interschimbă, și se schimbă și modul de parcursere (pi=0; pj=1 – avansează indicele j).

Se compară elementul din poziția i (3) cu elementul din poziția j (5). Deoarece 3 este mai mic decât 5, cele două valori nu se interschimbă, și se păstrează modul de parcursere (pi=0; pj=1 – avansează indicele j).

Se compară elementul din poziția i (3) cu elementul din poziția j (1). 3 fiind mai mare decât 1, cele două valori se interschimbă și se schimbă și modul de parcursere (avansează indicele i). Cei doi indici fiind egali, algoritmul se termină.

	i		j	
1	2	3	4	5
2	3	1	5	4

	i		j	
1	2	3	4	5
2	3	1	5	4

		i j		
1	2	3	4	5
2	1	3	5	4

Versiunea 2

```
void divizeaza(int s,int d,int &m)
{int pivot=x[s],i=s,j=d;
 while (i<j) {while(x[i]<pivot) i++;
             while(x[j]>pivot) j--;
             if (i<j) schimb(x[i],x[j]);}
             m=i;}
```

Inițial, cei doi indici i și j sunt inițializați cu extremitățile vectorului ($i=1; j=5$) și pivotul are valoarea 3

Elementul din poziția i (3) nu este mai mic decât pivotul; indicele i nu avansează ($i=1$). Elementul din poziția j (2) nu este mai mare decât pivotul; indicele j nu avansează ($j=5$). Valorile din pozițiile i și j se interschimbă.

Elementul din poziția i (2) este mai mic decât pivotul; indicele i avansează până la primul element mai mare decât pivotul ($i=2$). Elementul din poziția j (3) nu este mai mare decât pivotul; indicele j ($j=5$) nu avansează. Valorile din pozițiile i și j se interschimbă.

Elementul din poziția i (3) nu este mai mic decât pivotul; indicele i avansează la primul element mai mare decât pivotul ($i=4$). Elementul din poziția j (4) este mai mare decât pivotul; indicele j avansează până la primul element mai mic decât pivotul ($j=3$). Valorile din pozițiile i și j se interschimbă.

Elementul din poziția i (1) este mai mic decât pivotul; indicele i avansează la primul element mai mare decât pivotul ($i=4$). Elementul din poziția j (3) nu este mai mare decât pivotul; indicele j nu avansează ($j=3$). Indicele i fiind mai mare decât indicele j, algoritmul se termină.

	i			j
1	2	3	4	5
3	4	1	5	2

	i			j
1	2	3	4	5
2	4	1	5	3

	i			j
1	2	3	4	5
2	3	1	5	4

	i		j	
1	2	3	4	5
2	1	3	5	4

		j		i
1	2	3	4	5
2	1	3	5	4

Observație. În ambele cazuri algoritmul continuă cu divizarea vectorului în subvectorii cu indicii [1,2] și [4,5] și rearanjarea elementelor în cei doi subvectori.

Complexitatea algoritmului de sortare rapidă Pentru divizarea problemei în subprobleme se calculează mijlocul intervalului de indici și $O(D(n))=\Theta(1)$. Pentru combinarea soluțiilor se parcurge vectorul cu ajutorul celor doi indici, de la primul element până la ultimul element, și $O(D(n))=O(n \times \Theta(1))=O(n)$. Timpul de execuție este $T(n)=2 \times T(n/2)+n$. Considerând că $n=2^k$, rezultă:

$$\begin{aligned} T(n) &= T(2^k) + 2^k = 2 \times (T(2^{k-1}) + 2^{k-1}) + 2^k = 2 \times T(2^{k-1}) + 2^k + 2^k = \dots = \\ &2 \times 2 \times (T(2^{k-2}) + 2^{k-2}) + 2^k + 2^k = 2^2 \times (T(2^{k-2}) + 2^{k-2}) + 2^k + 2^k + 2^k = k \times 2^k = \log_2 n \times n. \end{aligned}$$

Ordinul de complexitate al algoritmului este $O(n \times \log_2 n)$.

1.4.5. Sortarea prin interclasare (MergeSort)

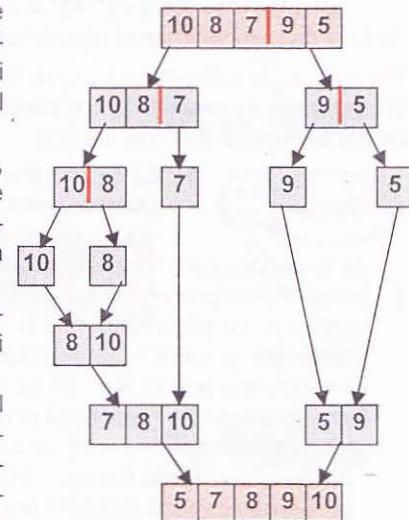
Algoritmul de interclasare se execută pe doi vectori, ordonați după același criteriu, pentru a obține un al treilea vector, care să conțină elementele primilor doi vectori, ordonate după același criteriu. Algoritmul de sortare prin interclasare se bazează pe observația că orice vector care conține un singur element este un vector sortat. Algoritmul de interclasare se poate folosi pentru sortarea unui vector cu ajutorul metodei **divide et impera**, astfel:

PAS1. Se descompune problema în subprobleme similare, prin împărțirea vectorului în doi subvectori, având multimea indicilor $[s, m]$ și $[m+1, d]$, unde m este indicele din mijlocul intervalului: $m = (s+d)/2$.

PAS2. Dacă subvectorul conține un singur element, atunci se consideră sortat (coresponde cazului de bază); altfel, se continuă descompunerea lui în subvectorii care au multimea indicilor $[s, m]$ și multimea indicilor $[m+1, d]$.

PAS3. Se combină soluțiile celor două subprobleme, prin interclasarea celor doi vectori sortați, obținându-se un vector sortat.

Vectorul care se sortează este x . Prin subprogramul **interclaseaza()** se realizează combinarea soluțiilor prin interclasarea subvectorului x , care are multimea indicilor $[s, m]$, cu subvectorul x , care are multimea indicilor $[m+1, d]$, în vectorul auxiliar v . Vectorul v se copiază în vectorul x , care are multimea indicilor $[s, d]$.



```

#include<iostream.h>
int x[100],n;
void divizeaza(int s,int d,int &m) {m=(s+d)/2;}
void interclaseaza(int s,int d,int m)
{int i=s,j=m+1,k=1,v[100];
 while (i<=m && j<=d)
 {if (x[i]<x[j]) {v[k]=x[i]; i++;}
  else {v[k]=x[j]; j++;}
  k++;}
 if (i<=m) while (i<=m) {v[k]=x[i]; i++; k++;}
 else while (j<=d) {v[k]=x[j]; j++; k++;}
 for (k=1,i=s;i<=d;k++,i++) x[i]=v[k];
}
void MergeSort(int s,int d)
{int m;
 if (s<d) {divizeaza(s,d,m);
 MergeSort(s,m);
 MergeSort(m+1,d);
 interclaseaza(s,d,m);}}
  
```

```
void main()
{int i; cout<<"n= ";cin>>n;
for(i=1;i<=n;i++) {cout<<"x["<<i<<"]= ";cin>>x[i];}
MergeSort(1,n);
cout<<"vectorul sortat"<<endl; for(i=1;i<=n;i++) cout<<x[i]<< " ";
```

Complexitatea algoritmului de sortare prin interclasare. Pentru divizarea problemei în subprobleme se calculează mijlocul intervalului de indici și $O(D(n))=O(1)$. Pentru combinarea soluțiilor se execută interclasarea a doi vectori și $O(D(n))=O(n)$. Timpul de execuție este $T(n)=2 \times T(n/2)+n$. Considerând că $n=2^k$, rezultă:

$$T(n)=T(2^k)+2^k = 2 \times (T(2^{k-1})+2^{k-1}) + 2^k = 2 \times T(2^{k-1}) + 2^k + 2^k = \dots = \\ 2 \times 2 \times (T(2^{k-2})+2^{k-2}) + 2^k + 2^k = 2^2 \times (T(2^{k-2})+2^{k-2}) + 2^k + 2^k + 2^k = k \times 2^k = \log_2 n \times n.$$

Ordinul de complexitate al algoritmului este $O(n \times \log_2 n)$.

Observație. În comparație cu algoritmii de sortare prin metoda selecției directe și prin metoda bulelor, care au ordinul de complexitate $O(n^2)$, algoritmii de sortare care folosesc strategia **divide et impera** sunt mai eficienți.

Temă



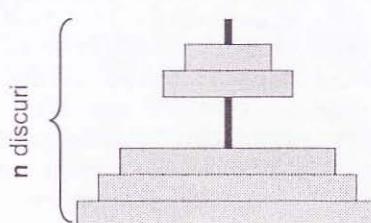
- Să se rearanjeze elementele unui vector astfel încât în vector să se găsească mai întâi numerele impare și apoi numerele pare. Se vor realiza două versiuni ale programului, câte una pentru fiecare metodă de sortare care folosește **strategia divide et impera**.
- Într-un fișier text sunt scrise următoarele informații: pe primul rând numărul de elevi din clasă – n , iar pe următoarele n rânduri următoarele informații despre un elev: numele, prenumele și mediile semestriale la disciplina informatică. În cadrul unui rând datele sunt separate prin spațiu. Să se scrie un program care să realizeze următoarele cerințe. Fiecare cerință va fi rezolvată printr-un subprogram. Ordonarea elementelor vectorului și căutarea unui elev în vector se vor face folosind algoritmii cei mai eficienți.
 - Se citesc datele din fișierul text și se calculează media anuală a fiecărui elev.
 - Se rearanjează datele în ordinea alfabetică a numelui și prenumelui elevilor.
 - Pentru numele și prenumele unui elev, citite de la tastatură, se afișează mediile semestriale și media anuală.
 - Se scriu informațiile obținute în urma prelucrărilor într-un alt fișier text.

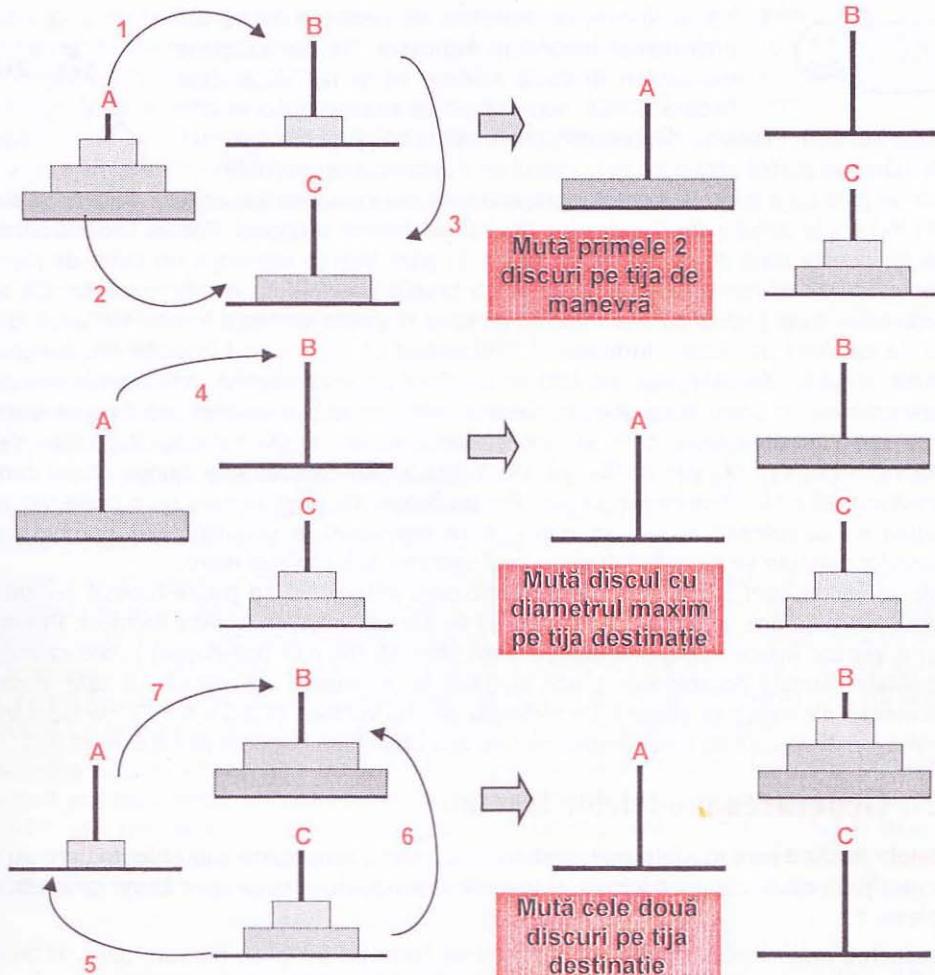
1.4.6. Problema turnurilor din Hanoi

Pe trei tije notate cu A, B și C se pot monta discuri perforate de diferite dimensiuni. Inițial pe tija A (tija sursă) sunt așezate unele peste altele n discuri în ordinea descrescătoare a diametrelor, iar celelalte tije sunt goale. Să se afișeze toate mutările care trebuie făcute ca discurile de pe tija A să fie aranjate pe tija B (tija destinație) în aceeași ordine în care erau pe tija A, folosind tija C ca tijă de manevră. O mutare nu se poate face decât cu un disc și el nu poate fi așezat pe o tijă decât peste un disc cu diametrul mai mare.

Să analizăm mutările care trebuie făcute pentru $n=3$:

- Se mută discul de pe tija A pe tija B (1).
- Se mută discul de pe tija A pe tija C (2).
- Se mută discul de pe tija B pe tija C (3).
- Se mută discul de pe tija A pe tija B (4).
- Se mută discul de pe tija C pe tija A (5).
- Se mută discul de pe tija C pe tija B (6).
- Se mută discul de pe tija A pe tija B (7).





Folosind metoda **divide et impera** problema initială va fi descompusă în subprobleme astfel:

- PAS1.** Se mută primele $n-1$ discuri de pe tija sursă pe tija de manevră.
- PAS2.** Se mută discul cu diametrul cel mai mare de pe tija sursă pe tija destinație.
- PAS3.** Se mută cele $n-1$ discuri de pe tija de manevră pe tija destinație.

```
#include<iostream.h>
void hanoi(int n, char a, char b, char c)
{if (n==1) cout<<"Mutarea: "<<a<<"->"<<b<<endl;
else
{hanoi(n-1,a,c,b);
 cout<<"Mutarea: "<<a<<"->"<<b<<endl;
 hanoi(n-1,c,b,a);}}
void main()
{int n; char a='A',b='B',c='C'; cout<<"n= "; cin>>n;
 hanoi(n,a,b,c);}
```

Temă

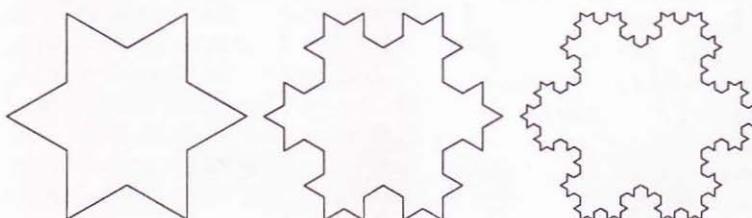
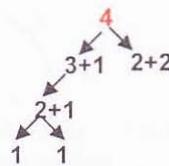
- Să se determine numărul de partiții distincte ale unui număr natural n . **Indicație.** Se descompune numărul n în două numere n_1 și n_2 , după care fiecare număr nou obținut se descompune în alte două numere. Procesul de descompunere continuă până când $n_1 < n_2$. Numărul de partiții este egal cu numărul de descompuneri obținute.
- Într-un parc cu o suprafață dreptunghiulară care are coordonatele colțului din stânga sus (X_1, Y_1) și ale colțului din dreapta jos (X_2, Y_2) se găsesc n copaci. Poziția fiecărui copac pe teren este dată de coordonatele (x_i, y_i) . În parc trebuie amenajat un teren de joacă pentru copii, de formă dreptunghiulară, cu laturile paralele cu laturile parcului. Să se determine dreptunghiul cu arie maximă pe care se poate amenaja terenul de joacă fără să se taie nici un copac. **Indicație.** Considerând că un copac împarte dreptunghiul inițial în patru dreptunghiuri cu laturile paralele cu dreptunghiul inițial, problema se descompune în patru subprobleme, fiecare subproblemă rezolvând cazul unuia dintre cele patru dreptunghiuri, care au coordonatele colțurilor $((X_1, Y_1); (x_i, Y_2))$, $((x_i, Y_1); (X_2, Y_2))$, $((X_1, Y_1); (X_2, y_i))$ și $((X_1, y_i); (X_2, Y_2))$. La cazul de bază se ajunge atunci când dreptunghiul nu conține nici un copac. Pentru fiecare dreptunghi care nu conține nici un copac se calculează aria și se memorează coordonatele colțurilor. Din combinarea soluțiilor obținute se identifică dreptunghiul care are aria cea mai mare.
- Într-un vector sunt memorate mai multe numere întregi. Să se plieze repetat vectorul, până când rămâne un singur element, și să se afișeze valoarea acestui element. **Plierea unui vector** înseamnă suprapunerea unei jumătăți (numită donatoare) peste cealaltă jumătate (numită receptoare). Dacă numărul de elemente ale vectorului este impar, elementul din mijloc se elimină. De exemplu, pentru vectorul $\{1, 2, 3, 4, 5, 6, 7\}$, considerând prima jumătate ca fiind cea receptoare, vectorul obținut după prima pliere este $\{5, 6, 7\}$.

1.4.7. Generarea modelelor fractale

Modelele fractale sunt modele matematice care descriu fenomene sau obiecte care au o structură periodică. Ele pot fi folosite și în grafică, pentru desenarea unor forme geometrice complexe.

În descrierea matematică a formei geometrice se pornește de la un **nucleu**, care, în urma unui proces iterativ, este generat succesiv la scări diferite. Numărul de iterații reprezintă ordinul curbei. Un exemplu clasic este forma geometrică cunoscută sub numele de **curba lui Koch** care se generează astfel:

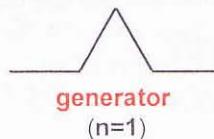
- PAS1.** Se pornește de la **nucleul** care este un triunghi echilateral, cu latura având lungimea finită L .
- PAS2.** Fiecare latură a triunghiului este împărțită în trei segmente egale. Segmentul din mijloc este eliminat și înlocuit cu un unghi care are laturile egale cu el.
- PAS3.** Pentru fiecare iterație se repetă Pasul 2, pentru fiecare segment al figurii obținute.



Se obține o construcție perfect regulată. După fiecare iterație cresc: numărul de unghiuri, numărul de laturi ale poligonului și perimetrul poligonului. Notăm cu p perimetrul nucleu (triunghiul echilateral de la care se pornește). În procesul de construcție a curbei, la fiecare iterație, fiecare latură L_i a triunghiului echilateral este împărțită în trei segmente de lungime egală ($L_i/3$) și este înlocuită cu 4 astfel de segmente ($4 \times L_i/3$). După n iterări, perimetrul poligonului va fi $p \times (4/3)^n$.

Pentru realizarea construcției se folosesc următoarele 3 elemente:

inițiator
($n=0$)



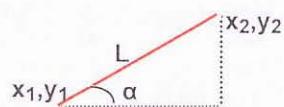
generator
($n=1$)



nucleu

Pentru construirea poligonului trebuie desenată fiecare latură. Pentru fiecare latură se cunosc: coordonatele x_1 și y_1 , lungimea segmentului L și unghiul α . Coordonatele x_2 și y_2 se pot determina:

$$x_2 = x_1 + L \cos(\alpha) \text{ și } y_2 = y_1 + L \sin(\alpha).$$



Pentru desenarea segmentului se folosește funcția **line()** implementată în biblioteca **graphics.h**: `line(x1, y1, x2, y2)`.

Desenarea poligonului înseamnă desenarea fiecărei laturi, care este determinată de coordonatele x_1 și y_1 , lungimea segmentului L și unghiul α . Problema se reduce la generarea datelor care caracterizează fiecare latură și la desenarea unei linii folosind aceste date. Generarea laturilor poligonului este un proces recursiv în care inițiatorul care are lungimea L este înlocuit cu generatorul. Generatorul este format din patru segmente, fiecare segment având lungimea $L/3$. Acest proces poate fi descompus în patru procese.

Folosind metoda **divide et impera** pentru a genera curba lui Koch de ordinul n , problema inițială va fi descompusă în patru subprobleme, astfel (subprogramul **Koch()**):

- PAS1.** Se generează primul segment, care este un segment cu lungimea $L/3$ și cu aceeași orientare cu al inițiatorului. El devine inițiatorul curbei lui Koch de ordinul $n-1$.
- PAS2.** Se generează al doilea segment, prin rotirea segmentului obținut cu 60° spre stânga (subprogramul **stanga()**). El devine inițiatorul curbei lui Koch de ordinul $n-1$.
- PAS3.** Se generează al treilea segment, prin rotirea segmentului obținut cu 120° spre dreapta (subprogramul **dreapta()**). El devine inițiatorul curbei lui Koch de ordinul $n-1$.
- PAS4.** Se generează al patrulea segment, prin rotirea segmentului obținut cu 60° spre stânga (subprogramul **stanga()**). El devine inițiatorul curbei lui Koch de ordinul $n-1$.

Procesul de descompunere se termină când se ajunge la ordinul 0 și se obține inițiatorul, care se va desena (subprogramul **deseneaza()**).

Pentru desenarea poligonului, se construiește nucleul (se generează cele trei laturi ale triunghiului echilateral de lungime L) și pentru fiecare latură se construiește curba lui Koch de ordinul n (subprogramul **nucleu_Koch()**).

```
#include<iostream.h>
#include<math.h>
#include<graphics.h>
int x,y;
float alfa;
void stanga(float unghi)
{alfa+=unghi*M_PI/180.;}
```

```

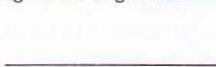
void dreapta(float unghi)
{alfa=unghi*M_PI/180.;}
void deseneaza(float L)
{x1=x; y1=y; x+=(int)(L *cos(alfa)); y+=(int)(L*sin(alfa));
line(x,y,x1,y1);}
void Koch(int n, float L)
{if (n==0) deseneaza(L);
else
{Koch(n-1,L/3); stanga(60);
Koch(n-1,L/3); dreapta(120);
Koch(n-1,L/3); stanga(60);
Koch(n-1,L/3);}}
void nucleu_Koch(int n,float L)
{Koch(n,L); dreapta(120);
Koch(n,L); dreapta(120);
Koch(n,L); dreapta(120);}
void main()
{int n,L;
cout<<"n= "; cin>>n; cout<<"L= "; cin>>L;
nucleu_Koch(n,L);}

```

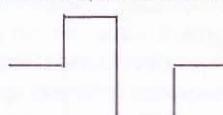
Temă

1. Desenați **praful lui Cantor**, care este generat astfel: un segment de lungimea L , care este paralel cu axa Ox , este împărțit în cinci segmente egale, din care se elimină segmentul din mijloc – și se repetă acest proces de n ori pentru fiecare segment rămas.

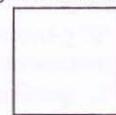
2. Desenați **curba lui Koch**, definită prin următoarele trei elemente, în care fiecare segment al generatorului are lungimea $L/4$, unde L este lungimea segmentului inițiator.



inițiator
($n=0$)



generator
($n=1$)



nucleu

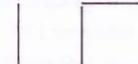
3. Desenați **curba dragonului**, definită prin următoarele trei elemente, în care fiecare segment al generatorului are lungimea $L/\sqrt{2}$, unde L este lungimea segmentului inițiator.



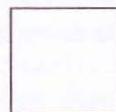
inițiator
($n=0$)



generator
($n=1$)



curba de
ordinul $n=2$



nucleu

4. Să se deseneze figura geometrică obținută astfel: se desenează un pătrat cu latura L , se desenează un pătrat care unește mijloacele laturilor pătratului inițial și se repetă acest proces de n ori pentru fiecare pătrat obținut.
5. Să se deseneze **covorul lui Sierpinski**, astfel: se desenează un pătrat cu latura L , se împarte acest pătrat în 9 pătrate egale, se hașurează pătratul din mijloc și se repetă acest proces de n ori pentru fiecare pătrat nehașurat.

1. 5. Metoda greedy

1.5.1. Descrierea metodei greedy

Metoda greedy se poate folosi pentru problemele în care, dându-se o **multime finită A**, trebuie determinată o multime $S \subseteq A$ care să îndeplinească anumite **condiții**. Metoda furnizează o singură soluție reprezentată prin elementele multimei S. Ca și în cazul metodei backtracking, soluția problemei este dată de un vector $S = \{x_1, x_2, \dots, x_n\}$ ale cărui elemente aparțin însă unei singure multimi A. Spre deosebire de metoda backtracking, metoda Greedy nu găsește decât o singură soluție și, în general, această soluție este **soluția optimă**.

Studiu de caz

Scop: identificarea problemelor în care **soluția optimă** este o **submultime inclusă** într-o multime dată, care trebuie să îndeplinească anumite condiții.

Enunțul problemei 1: Să se **repartizeze optim o resursă** (de exemplu, o sală de spectacole, o sală de conferințe, o sală de sport) mai multor activități (spectacole, prezentări de produse, respectiv meciuri) care concurează pentru a obține resursa respectivă.

Multimea A este formată din cele n activități. Fiecare activitate i ($1 \leq i \leq n$) are un **temp de începere** t_i și un **temp de terminare** f_i , unde $t_i < f_i$ și ocupă resursa în intervalul de temp $[t_i, f_i]$. Două activități i și j sunt compatibile dacă intervalele lor de ocupare $[t_i, f_i]$ și $[t_j, f_j]$ sunt disjuncte, adică dacă $f_i \leq t_j$ sau dacă $f_j \leq t_i$. Cerința problemei este de a selecta o multime maximală de activități compatibile. În acest caz, multimea S este formată din activitățile care vor folosi resursa, iar condiția pe care trebuie să o îndeplinească elementele multimei S este ca ele să fie activități compatibile. În plus, pentru ca repartizarea resursei să fie optimă, trebuie ca multimea S să conțină maximul de elemente care îndeplinesc această condiție.

Enunțul problemei 2: Să se **ocupe optim un mijloc de transport** (de exemplu, un rucsac, un autocamion) care are o capacitate maximă de ocupare (care poate transporta o greutate maximă G) cu n obiecte, fiecare obiect având greutatea g_i și un profit obținut în urma transportului c_i , iar din fiecare obiect putând să se ia o fracțiune $x_i \in [0, 1]$.

Multimea A este formată din cele n obiecte. Fiecare obiect i ($1 \leq i \leq n$) are o **eficiență a transportului** e_i care reprezintă profitul pentru o unitatea de greutate. Cerința problemei este de a selecta o multime de obiecte astfel încât eficiența transportului să fie maximă. Multimea S este formată din obiectele care vor ocupa mijlocul de transport, iar condiția pe care trebuie să o îndeplinească elementele multimei S este ca, prin contribuția adusă de fiecare obiect la eficiența transportului, să se obțină o eficiență maximă, iar greutatea obiectelor selectate să fie egală cu greutatea maximă a transportului.

Enunțul problemei 3: Pentru două multimi de numere întregi nenule: C cu n elemente, $C = \{c_1, c_2, \dots, c_n\}$ și A cu m elemente, $A = \{a_1, a_2, \dots, a_m\}$, și $n \leq m$, să se selecțeze o submultime de n numere din multimea A, astfel încât **expresia**:

$$E = c_1 \times x_1 + c_2 \times x_2 + \dots + c_n \times x_n$$

în care $x_i \in A$, să aibă valoarea maximă.

În acest caz, multimea S = { x_1, x_2, \dots, x_n }, în care $x_i \in A$, trebuie să îndeplinească condiția:

$$E = c_1 \times x_1 + c_2 \times x_2 + \dots + c_n \times x_n = E_{\max}$$

Criteriul de alegere a elementelor x_i este următorul: dacă în multimea A mai există elemente care au același semn cu coeficientul c_i , se va alege elementul aj pentru care termenul $c_i \times x_j$

are valoarea maximă (deoarece acest termen se adună la valoarea expresiei), iar dacă în mulțimea A nu mai există elemente care au același semn cu coeficientul c_i , se va alege elementul a_j pentru care termenul $c_i \times x_j$ are valoarea minimă (deoarece acest termen se scade din valoarea expresiei).

Enunțul problemei 4: Să se plătească o sumă s cu un număr minim de bancnote cu valori date. Se consideră că din fiecare tip de bancnotă se poate folosi un număr nelimitat de bancnote, iar pentru ca problema să aibă soluție, vom considera că există și bancnote cu valoarea 1.

Mulțimea A este formată din cele n valori distincte ale bancnotelor. Cerința problemei este de a selecta o mulțime minimă de bancnote pentru plata sumei s . În acest caz, mulțimea S este formată din valorile cu care se va face plata, iar condiția pe care trebuie să o îndeplinească elementele mulțimii S este ca, prin adunarea sumelor parțiale plătite cu bancnotele cu valorile alese, să se obțină suma de plată s . Pentru ca numărul de bancnote să fie cât mai mic, trebuie să se caute ca plata să se facă în bancnote cu valori cât mai mari.



Metoda **greedy** construiește soluția prin selectarea, dintr-o mulțime de elemente, a elementelor care îndeplinesc o anumită condiție. Pentru ca elementele care se selectează să aparțină **soluției optime**, la pasul k se alege **candidatul optim** pentru elementul x_k al soluției.

Spre deosebire de metoda backtracking, la metoda greedy, alegerea elementului x_k al soluției este irevocabilă (nu se mai poate reveni asupra alegerii făcute).

Metoda greedy poate duce la obținerea soluției optime în cazul problemelor care au proprietatea de optim local, adică soluția optimă a problemei cu dimensiunea n a datelor de intrare conține soluțiile optime ale subproblemelor similare cu problema initială, dar de dimensiune mai mică. Metoda greedy se mai numește și **metoda optimului local**.

Pașii algoritmului greedy sunt:

- PAS1.** Se inițializează mulțimea S cu mulțimea vidă: $S \leftarrow \emptyset$.
- PAS2.** Cât timp S nu este soluție a problemei și $A \neq \emptyset$, execută:
 - PAS3.** Se alege din mulțimea A elementul a care este **candidatul optim** al soluției.
 - PAS4.** Se elimină elementul a din mulțimea A.
 - PAS5.** Dacă el poate fi element al soluției, atunci elementul a se adaugă la mulțimea S . Se revine la Pasul 2.
- PAS6.** Dacă mulțimea S este soluția problemei, atunci se afișează soluția; altfel, se afișează mesajul "Nu s-a găsit soluție"

Algoritmul greedy este un **algoritm iterativ**, care determină soluția optimă a problemei în urma unor succesiuni de alegeri care reduc dimensiunea problemei respective: se alege elementul x_1 al soluției, apoi elementul x_2 al soluției, și.a.m.d. Altfel spus, elementul x_k al soluției este determinat prin alegerea, din elementele rămase în mulțimea A, a candidatului optim pentru elementul x_k al soluției, iar determinarea următoarele $n-k$ elemente ale soluției (rezolvarea subproblemelor) se face numai după ce a fost determinat elementul x_k al soluției. Alegerea făcută pentru elementul x_k al soluției poate depinde de alegerile făcute pentru determinarea primelor $k-1$ elemente ale soluției, dar nu depinde de alegerile ulterioare.

Pentru ca algoritmul greedy să conducă la obținerea soluției optime, trebuie să fie îndeplinite două condiții:

1. Alegerea optimului local pentru fiecare element al soluției duce la alegerea soluției optime globale.

2. Soluția optimă a problemei conține soluțiile optime ale subproblemelor.

Aceasta înseamnă că, pentru a fi siguri că algoritmul greedy construiește soluția optimă a problemei, trebuie să se demonstreze că sunt îndeplinite cele două condiții.

În rezolvarea multor probleme folosind strategia greedy, pentru a alege optimul local care să ducă la alegerea soluției optime globale, **multimea A este ordonată după criteriul candidatului optim**. Ordonarea mulțimii după criteriul candidatului optim înseamnă reanjarea elementelor mulțimii A astfel încât, după extragerea unui element, următorul element din mulțimea A să reprezinte elementul care este cel mai îndreptățit să fie ales ca element al mulțimii S. De exemplu:

- În cazul **repartizării optime a unei resurse**, o activitate nu poate începe decât dacă se termină cea anterioară. A repartiza optim resursa înseamnă a organiza cât mai multe activități. Din mulțimea activităților se alege întotdeauna activitatea care are timpul de terminare cât mai mic. În acest caz, candidatul optim este activitatea care are timpul de terminare cel mai mic, iar ordonarea activităților după criteriul candidatului optim înseamnă ordonarea crescătoare după timpul de terminare.
- În cazul **ocupării optime a mijlocului de transport**, trebuie ca eficiența transportului să fie maximă. Din mulțimea obiectelor, se aleg obiectele care au eficiență de transport cât mai mare. În acest caz, candidatul optim este obiectul care are eficiența transportului cea mai mare, iar ordonarea obiectelor după criteriul candidatului optim înseamnă ordonarea descrescătoare după eficiența transportului.
- În cazul **plății unei sume cu un număr minim de bancnote cu valoare dată**, ca să se folosească cât mai puține bancnote, trebuie ca plata să se facă în cât mai multe bancnote cu valoare foarte mare. În acest caz, candidatul optim este bancnota cu valoarea cea mai mare, iar ordonarea după criteriul candidatului optim înseamnă ordonarea descrescătoare după valoarea bancnotelor.

1.5.2. Implementarea metodei greedy

Pentru implementarea metodei greedy se vor folosi două subprograme:

- subprogramul **sort()** – care ordonează mulțimea A după criteriul candidatului optim;
- subprogramul **greedy()** – care implementează metoda propriu-zisă.

În exemplele următoare, datele de intrare vor fi citite dintr-un fișier text.

Exemplul 1. Problema planificării optime a activităților

Se folosesc următoarele date și structuri de date:

- Pentru numărul de elemente ale celor două mulțimi se folosesc variabilele **n** – numărul de activități (numărul de elemente ale mulțimii A) și **m** – numărul de activități selectate pentru a forma soluția problemei (numărul de elemente ale mulțimii S).
- Pentru mulțimea activităților (mulțimea A) se folosește vectorul **a**, ale cărui elemente sunt de tip **înregistrare activitate** – care conține trei câmpuri, **x** și **y** pentru ora la care începe activitatea, respectiv ora la care se termină activitatea – și **k** pentru numărul activității (este folosit ca un identificator al activității).
- Soluția (mulțimea S) va fi memorată în vectorul **s**. În fiecare element al vectorului **s** se memorează indicele elementului selectat din vectorul **a**, după ce a fost sortat după criteriul candidatului optim.

Se folosesc următoarele subprograme:

- Subprogramul **citeste()** – pentru citirea datelor din fișierul text. În fișierul text, pe primul rând este scris numărul de activități **n**, iar pe următoarele **n** rânduri – perechi de numere întregi, separate prin spațiu, care reprezintă ora de început și ora de sfârșit ale unei activități. Fiecarei activități i se atribuie ca identificator numărul de ordine de la citirea din fișier. În urma execuției acestui subprogram, se creează vectorul activităților – **a**.
- Subprogramul **sort()** – sortează vectorul **a** crescător după timpul de terminare a activităților (câmpul **y**).
- Subprogramul **greedy()** – implementează strategia greedy pentru această problemă.
- Subprogramul **afiseaza()** – afișează soluția problemei folosind informațiile din vectorul **s** – indicele fiecărei activități selectate.

Strategia greedy este implementată astfel:

- PAS1.** Se inițializează vectorul **s** cu indicele primului element din vectorul **a** (se selezează, ca primă activitate, activitatea care are ora de terminare cea mai mică).
- PAS2.** Pentru următoarele **n-1** elemente ale vectorului **a**, execută
- PAS3.** Dacă ora la care începe activitatea din elementul curent al vectorului **a** este mai mare sau egală cu ora la care se termină ultima activitate adăugată la soluție (în vectorul **s**), atunci activitatea este adăugată la soluție.

Mulțimea activităților

Activitatea	1	2	3	4	5	6	7	8
Ora de începere	9	12	8	10	16	14	20	19
Ora de terminare	11	13	10	12	18	16	22	21

Mulțimea activităților – după ce a fost sortată

Activitatea	3	1	4	2	6	5	8	7
Ora de începere	8	9	10	12	14	16	19	20
Ora de terminare	10	11	12	13	16	18	21	22

Soluția problemei

Activitatea	3	4	2	6	5	8
Ora de începere	8	10	12	14	16	19
Ora de terminare	10	12	13	16	18	21

Programul este:

```
#include<iostream.h>
struct activitate {int x,y,k;};
activitate a[20];
int n,m,s[20];
fstream f("greedy1.txt",ios::in);
void citeste()
{int i; f>>n;
 for(i=1;i<=n;i++) {f>>a[i].x>>a[i].y; a[i].k=i;} f.close();}
void sort()
{int i,j; activitate aux;
 for (i=1;i<n;i++)
  for (j=i+1;j<=n;j++)
   if (a[i].y>a[j].y) {aux=a[i]; a[i]=a[j]; a[j]=aux;}}
```

```

void greedy()
{int i,j; s[1]=1; j=1;
 for (i=2;i<=n;i++)
    if (a[i].x>=a[s[j]].y) {j++; s[j]=i;}
 m=j;}
void afiseaza()
{cout<<"Planificarea activitatilor: "<<endl;
 for (int i=1;i<=m;i++)
    cout<<"Activitatea "<<a[s[i]].k<<" incepe la ora ";
    cout<<a[s[i]].x <<" si se termina la ora "<<a[s[i]].y<<endl;}
void main()
{citeste(); sort(); greedy(); afiseaza();}

```

Temă

Într-o sală de spectacole trebuie planificate n spectacole care au loc în aceeași zi, fiecare spectacol având o oră la care trebuie să înceapă și o durată de desfășurare. Scrieți un program care să planifice optim ocuparea sălii de spectacole.

Exemplul 2. Problema **ocupării optime a mijlocului de transport** (problema **rucsacului**).

Se folosesc următoarele date și structuri de date:

- Pentru numărul de elemente ale celor două mulțimi se folosesc variabilele n – numărul de obiecte (numărul de elemente ale mulțimii A) și m – numărul de obiecte selectate pentru a forma soluția problemei (numărul de elemente ale mulțimii S).
- Pentru greutatea maximă care se poate transporta, se folosește variabila G , iar pentru greutatea obiectelor care se mai pot selecta, se folosește variabila Gr (greutatea rămasă). Variabila Gr este inițializată cu valoarea G (inițial, greutatea obiectelor care se pot selecta este greutatea maximă de transport), iar după ce s-a construit soluția, variabila Gr are valoarea 0 (nu se mai pot selecta obiecte, deoarece a fost ocupată toată capacitatea de transport).
- Pentru mulțimea obiectelor (A) se folosește vectorul a , ale cărui elemente sunt de tip înregistrare **obiect**, care conține patru câmpuri: g – greutatea obiectului, c – profitul obținut în urma transportului, e – eficiența transportului și k – numărul obiectului (este folosit ca un identificator al obiectului).
- Soluția (mulțimea S) va fi memorată în vectorii s și x . În fiecare element al vectorului s se memorează indicele obiectului selectat din vectorul a , după ce a fost sortat după criteriul candidatului optim, iar în elementul din vectorul x care are același indice cu cel din vectorul s se memorează fractiunea de cantitate care se va lua din obiectul selectat. Elementele vectorului x sunt inițializate cu valoarea 0 (vectorul este declarat global).

Mulțimea obiectelor – G=20

Obiectul	1	2	3	4	5
Greutatea	5	4	4	8	10
Profitul	10	20	10	10	22
Eficiență	2	5	2,5	1,25	2,2

Mulțimea obiectelor după ce a fost sortată

Obiectul	2	3	5	1	4
Greutatea	4	4	10	5	8
Profitul	20	10	22	10	10
Eficiență	5	2,5	2,2	2	1,25

Solutia problemei

Obiectul	2	3	5	1	4
Greutatea	4	4	10	5	8
Fracțiunea	1	1	1	0,2	0
Greutatea	4	4	10	2	0

Se folosesc următoarele subprograme:

- Subprogramul **citeste()** – pentru citirea datelor din fișierul text. În fișierul text, pe primul rând este scris numărul de obiecte **n** și greutatea maximă a transportului **G**, iar pe următoarele **n** rânduri – perechi de numere reale, separate prin spațiu, care reprezintă greutatea și profitul transportului unui obiect. Fiecare obiect î se atribuie, ca identificator, numărul de ordine de la citirea din fișier. Pentru fiecare obiect se calculează eficiența transportului. În urma execuției acestui subprogram se creează vectorul obiectelor, **a**.
- Subprogramul **sort()** – sortează vectorul **a**, crescător după eficiența transportului obiectelor (câmpul **e**).
- Subprogramul **greedy()** – implementează strategia greedy pentru această problemă.
- Subprogramul **afiseaza()** – afișează soluția problemei folosind informațiile din vectorii **s** și **x** – indicele fiecărui obiect selectat și fractiunea din greutate care se va transporta.

Strategia **greedy** este implementată astfel:

- PAS1.** Se initializează greutatea **Gr** cu valoarea **G** și selectarea obiectelor începe cu obiectul cu cea mai mare eficiență a transportului (primul obiect din vectorul **a**).
- PAS2.** Cât timp mai există obiecte care nu au fost selectate și greutatea obiectelor selectate nu este greutatea maximă, **execută**
- PAS3.** Dacă greutatea obiectului din elementul curent al vectorului **a** este mai mică sau egală cu greutatea rămasă, **Gr**, atunci obiectul este adăugat la soluție, luându-se întreaga cantitate disponibilă din obiect; **altfel**, se ia din obiect fractiunea egală cu greutatea rămasă **Gr**. Se actualizează greutatea **Gr** diminuând-o cu greutatea obiectului care a fost adăugat în rucsac.

Programul este:

```
#include<iostream.h>
struct obiect {int k;
               float g,c,e;};
obiect a[20];
int n,m,s[20];
float G,Gr,x[20];
fstream f("greedy2.txt",ios::in);
void citeste()
{int i; f>>n>>G;
 for(i=1;i<=n;i++)
  {f>>a[i].g>>a[i].c; a[i].k=i; a[i].e=a[i].c/a[i].g;} f.close();}
void sort()
{int i,j; obiect aux;
 for (i=1;i<n;i++)
  for (j=i+1;j<=n;j++)
   if (a[i].e<a[j].e) {aux=a[i]; a[i]=a[j]; a[j]=aux;}}
void greedy()
{int i,j=0; Gr=G;
 for (i=1;i<=n && Gr!=0;i++)
  if (Gr>a[i].g) {j++; s[j]=i; x[j]=1; Gr-=a[i].g;}
   else {j++; s[j]=i; x[j]=Gr/a[i].g; Gr=0;}
m=j;}
void afiseaza()
{for (int i=1;i<=m;i++)
```

```

    {cout<<"Obiectul "<<a[s[i]].k<<" in cantitatea ";
    cout<<x[i]*a[s[i]].g<<endl;}
void main()
{citere(); sort(); greedy(); afiseaza();}

```

Observație. În problema rucsacului există două situații:

1. Obiectele pot fi fracționate – problema **continuă** a rucsacului;
2. Obiectele nu pot fi fracționate – problema **discretă** a rucsacului.

Algoritmul greedy furnizează **soluția optimă** numai în cazul problemei **continuă a rucsacului**. De exemplu, pentru patru obiecte: a (20 kg, 110 lei), b (9 kg, 45 lei), c (9 kg, 45 lei) și d (6 kg, 30 lei) și greutatea maximă de transport 25 kg:

- Soluția greedy: obiectul **a** – profitul transportului = 110;
 → Soluția optimă: obiectele **b, c și d** – profitul transportului = 120.



Într-un rucsac se pot transporta maximum **G** kg. Există **n** obiecte, fiecare obiect având greutatea **gi** și valoarea **vi**. Obiectele pot fi fracționate. Scrieți un program care să găsească obiectele care trebuie transportate în rucsac, astfel încât să se obțină cea mai valoroasă încărcătură.

Exemplul 3. Problema expresiei de valoare maximă.

Se folosesc următoarele date și structuri de date:

- Pentru mulțimea coeficientilor **C** se folosește vectorul **c** cu **n** elemente, iar pentru mulțimea numerelor **A** se folosește vectorul **a** cu **m** elemente. Elementele vectorilor sunt de tip **int**.
- Pentru calcularea valorii expresiei se folosește variabila **exp**, care este inițializată cu valoarea 0 (este variabilă globală), iar pentru a număra elementele selectate din mulțimea **A** se folosește variabila **k**, care este inițializată cu valoarea 0.

Se folosesc următoarele subprograme:

- Subprogramul **citere()** – pentru citirea datelor din fișierul text. În fișierul text, pe primul rând sunt scrise două numere întregi, **n** și **m**, care reprezintă numărul de elemente ale celor două mulțimi, pe rândul al doilea cei **n** coeficienți, iar pe rândul al treilea cele **m** numere întregi din mulțimea **A**. Valorile numerice de pe un rând sunt separate prin spațiu. În urma execuției acestui subprogram se creează vectorii **c** și **a**.
- Subprogramul **sort()** – se folosește pentru a sorta crescător cei doi vectori **c** și **a**.
- Subprogramul **greedy()** – implementează strategia greedy pentru această problemă și afișează soluția în timp ce o construiește.

Strategia **greedy** este implementată astfel:

- PAS1.** Se calculează termenii pentru care valorile coeficientului **ci** și ale numărului **aj** sunt pozitive. Pentru ca expresia să fie maximă, parcurgerea celor doi vectori se face de la ultimul element către primul element (de la cele mai mari numere pozitive, către cele mai mici numere pozitive). Se memorează, în variabila **p**, indicele elementului din vectorul **c** care urma să fie selectat la terminarea parcurgerii, iar în variabila **r** – indicele elementului din vectorul **a** care urma să fie selectat la terminarea parcurgerii.
- PAS2.** Se calculează termenii pentru care valorile coeficientului **ci** și ale numărului **aj** sunt negative. Pentru ca expresia să fie maximă, parcurgerea celor doi vectori se face de la primul element către ultimul element (de la cele mai mari numere negative, către cele mai mici numere negative).

PAS3. Coeficientii rămași pot fi numai pozitivi sau numai negativi. În cazul în care sunt numai negativi, în mulțimea A au rămas numai numere pozitive. Termenii care se calculează fiind negativi, pentru ca expresia să fie maximă vor trebui să aibă valoarea cât mai mică și vectorii se vor parcurge în continuare, către ultimul element (coeficientii sunt numere negative din ce în ce mai mici, iar numerele din vectorul a sunt numere pozitive din ce în ce mai mari). În cazul în care coeficientii rămași sunt numai pozitivi, în mulțimea A au rămas numai numere negative. Termenii care se calculează fiind negativi, pentru ca expresia să fie maximă vor trebui să aibă valoarea cât mai mică și vectorii se vor parcurge astfel: vectorul c începând de la elementul p către primul element, iar vectorul a începând de la elementul r către primul element (coeficientii sunt numere pozitive din ce în ce mai mici, iar numerele din vectorul a sunt numere negative din ce în ce mai mari).

Programul este:

```
#include<fstream.h>
int n,m,p,r,c[20],a[20],exp;
fstream f("greedy3.txt",ios::in);
void citeste()
{int i; f>>n>>m;
for(i=1;i<=n;i++) f>>c[i];
for(i=1;i<=m;i++) f>>a[i];
f.close();}
void sort(int v[], int n)
{int i,j,aux;
for (i=1;i<n;i++)
    for (j=i+1;j<=n;j++)
        if (v[i]>v[j]) {aux=v[i]; v[i]=v[j]; v[j]=aux;}}
void greedy()
{int i=n,j=m,k=0; //indici: i-vector c, j-vector a, k-vector b
cout<<"E= ";
while (c[i]>0 && a[j]>0 && k<n)
{k++; cout<<"("<<c[i]<<")*(";<<a[j]<<")+" ;
exp+=c[i]*a[j]; i--; j--;}
p=j; i=1; j=1;
while (c[i]<0 && a[j]<0 && k<n)
{k++; cout<<"("<<c[i]<<")*(";<<a[j]<<")+" ;
exp+=c[i]*a[j]; i++; j++;}
if (c[i]<0)
while (k<n)
```

$$n=5 \text{ și } m=7 \rightarrow E_{\max} = 97$$

Indici	1	2	3	4	5	6	7
c _i	-2	-1	3	4	5		
a _i	-5	-4	-1	2	5	7	8
s _i	-5	-4	5	7	8		

$$n=6 \text{ și } m=7 \rightarrow E_{\max} = 77$$

Indici	1	2	3	4	5	6	7
c _i	-5	-4	-3	-1	4	5	
a _i	-5	-4	1	2	3	4	5
s _i	-5	-4	1	2	4	6	

$$n=6 \text{ și } m=7 \rightarrow E_{\max} = 21$$

Indici	1	2	3	4	5	6	7
c _i	-2	-1	3	4	5	5	
a _i	-6	-5	-4	-3	-1	1	2
s _i	-6	-5	-3	-1	1	2	

```

{k++; cout<< "("<<c[i]<<") * ("<<a[j]<<") + ";
exp+=c[i]*a[j]; i++; j++;}
else
{i=p;j=r;
while (k<n)
{k++; cout<< "("<<c[i]<<") * ("<<a[j]<<") + ";
exp+=c[i]*a[j]; i--; j--;}
cout<<'b'<< " = "<<exp;}
void main()
{citate(); sort(c,n); sort(a,m); greedy();}

```

Justificarea criteriului candidatului optim Considerăm cazul în care coeficienții sunt numere întregi pozitive care au fost ordonate crescător și există cel puțin n numere pozitive în mulțimea A. Conform criteriului candidatului optim, mulțimea A a fost ordonată crescător. Dacă $C=\{c_1, c_2, \dots, c_n\}$ cu $c_i \leq c_j$, pentru $i < j$ și $A=\{a_1, a_2, \dots, a_m\}$ cu $a_i \leq a_j$, pentru $i < j$, atunci $S=\{x_1, x_2, \dots, x_n\}$ cu $x_i \leq x_j$, pentru $i < j$ și $x_n=a_m, x_{n-1}=a_{m-1}, \dots, x_1=a_{m-n+1}$. Este evident că pentru a obține valoarea maximă a expresiei trebuie să se aleagă cele mai mari n numere din mulțimea A. Trebuie să demonstreazăm dacă este corect modul în care s-a construit mulțimea S cu aceste numere. Pentru alegerea făcută, expresia are valoarea E_1 .

$$E_1 = c_1 \times x_1 + c_2 \times x_2 + \dots + c_i \times x_i + \dots + c_j \times x_j + \dots + c_n \times x_n$$

Să presupunem că există o expresie E_2 a cărei valoare este mai mare decât a expresiei E_1 ($E_2 > E_1$). În cel mai bun caz, în expresie ordinea numerelor din mulțimea S diferă numai prin valorile din pozițiile i și j , cu $i < j$ și $c_i < c_j$ și $x_i < x_j$.

$$E_2 = c_1 \times x_1 + c_2 \times x_2 + \dots + c_i \times x_j + \dots + c_j \times x_i + \dots + c_n \times x_n$$

Conform presupunerii făcute, înseamnă că $E_2 - E_1 > 0$. Dar,

$$E_2 - E_1 = c_j \times (x_j - x_i) + c_i \times (x_i - x_j) = (c_j - c_i) \times (x_i - x_j) < 0 \text{ și } E_2 < E_1$$

ceea ce contrazice ipoteza de la care s-a plecat. În mod analog se justifică și alegerea elementelor mulțimii S atunci când cei n coeficienți sunt negativi și în mulțimea A există cel puțin n numere negative.



Tema

1. Justificați criteriul priorității de alegere pentru cazul în care coeficienții sunt pozitivi și în mulțimea A nu există decât numere negative.
2. Justificați criteriul priorității de alegere pentru cazul în care coeficienții sunt negativi și în mulțimea A nu există decât numere pozitive.
3. Refațeți programul astfel încât să se determine **valoarea minimă** a expresiei.
4. Fiind dată o mulțime A cu n numere reale, să se determine o submulțime a sa care are proprietatea că suma elementelor submulțimii este maximă. **Indicație.** Submulțimea va fi formată din toate numerele pozitive, iar dacă nu există numere pozitive, din cel mai mare număr din mulțime.
5. Fiind date mulțimea de numere reale $A = \{a_1, a_2, \dots, a_m\}$ și n , gradul unui polinom, cu $n < m$, să se selecteze, din mulțimea A, o submulțime de $n+1$ numere pentru coeficienții polinomului, astfel încât valoarea polinomului $P(x)$ într-un punct x precizat să fie maximă.
6. La un cabinet medical, în sala de așteptare, se găsesc n pacienți. Timpul necesar pentru consultarea unui pacient i este t_i . Cunoscând numărul de pacienți n și timpul necesar pentru fiecare consultație t_i , scrieți un program care să afișeze ordinea de tratare a pacienților, astfel încât timpul total de așteptare al pacienților să fie minim. **Indicație.** Deoarece timpul de așteptare al unui pacient este egal cu suma timpilor de consultăție ai pacienților tratați înaintea sa, ordonarea pacienților după criteriul

candidatului optim înseamnă ordonarea crescătoare după timpul de consultăție. Demonstrați că soluția obținută astfel este soluția optimă.

Exemplul 4. Problema plății unei sume cu un număr minim de bancnote cu valori date.

Se folosesc următoarele date și structuri de date:

- Pentru valorile bancnotelor se folosește vectorul **a** cu **n** elemente (**n** reprezintă numărul de tipuri de bancnote).
- Pentru suma care trebuie plătită se folosește variabila **s**.

Se folosesc următoarele subprograme:

$$n=4 \text{ și } S=147 \rightarrow \text{Număr total bancnote} = 9$$

- Subprogramul **citeste()** – pentru citirea datelor din fișierul text. În fișierul text, pe primul rând sunt scrise două numere întregi, **n** și **s**, iar pe rândul următor cele **n** valori ale bancnotelor. Valorile numerice de pe fiecare rând sunt separate prin spațiu.
- Subprogramul **sort()** – se folosește pentru a sorta descrescător vectorul **a**.
- Subprogramul **greedy()** – implementează strategia greedy pentru această problemă și afișează soluția în timp ce o construiește.

Valoare bancnotă	50	10	5	1
Număr de bancnote	2	4	1	2

Strategia **greedy** este implementată astfel:

PAS1. Selectarea bancnotelor începe cu bancnota cu valoarea cea mai mare (prima valoare din vectorul **a**).

PAS2. Cât timp nu a fost plătită toată suma, execută

PAS3. Dacă o parte din suma rămasă poate fi plătită în bancnote care au valoarea elementului curent din vectorul **a**, atunci se determină numărul maxim de bancnote cu care se face plata (câtul dintre suma rămasă și valoarea bancnotei) și se calculează restul din sumă care mai rămâne de plătit.

PAS4. Se trece la următoarea valoare a bancnotei și se revine la Pasul 2.

```
#include<iostream.h>
int n,m,S,a[20];
fstream f("greedy4.txt",ios::in);
void citeste()
{int i; f>>n>>S;
 for(i=1;i<=n;i++) f>>a[i]; f.close();}
void sort()
{int i,j,aux;
 for (i=1;i<n;i++)
    for (j=i+1;j<=n;j++)
        if (a[i]<a[j]) {aux=a[i]; a[i]=a[j]; a[j]=aux;}}
void greedy()
{int i=1;
 while (S!=0)
 {if (S/a[i]!=0)
     {cout<<S/a[i]<<" bancnote cu valoarea "<<a[i]<<endl;
      S=S%a[i];}
     i++;}}
void main()
{citeste(); sort(n); greedy();}
```

Observație. În plata sumei **S** cu bancnote de valori date pot să apară două cazuri (presupunând că în moneda respectivă există bancnote cu valorile precizate mai jos):

1. Suma este 48 și bancnotele au valorile 15, 5 și 4 ($n=3$). Algoritmul greedy nu va găsi nici o soluție, deoarece va alege 3 bancnote cu valoarea 15, rămânând un rest de sumă cu valoarea de 3, pentru care nu mai există bancnote ca să se plătească. În realitate, problema are o soluție: 2 bancnote cu valoarea 15, 2 bancnote cu valoarea 5, și 2 bancnote cu valoarea 8.
2. Suma este 22 și bancnotele au valorile 9, 7, 5 și 1 ($n=4$). Algoritmul greedy nu va găsi soluția optimă, deoarece va alege 2 bancnote cu valoarea 9, și 4 bancnote cu valoarea 1, în total 6 bancnote. Soluția optimă este de a plăti cu 4 bancnote: 3 bancnote cu valoarea 7 și o bancnotă cu valoarea 1.

Metoda greedy care nu furnizează soluția optimă a problemei se numește **metoda euristică greedy**.

 Ce relație trebuie să existe între valorile bancnotelor pentru ca soluția greedy să furnizeze soluția optimă?

Complexitatea algoritmului greedy

Algoritmul greedy propriu-zis are complexitatea $O(n)$, deoarece se parcurg cele n elemente ale mulțimii A pentru a alege fiecare dintre cele m elemente ale soluției. Algoritmul de sortare folosit pentru aranjarea elementelor mulțimii A după criteriul candidatului optim are complexitatea $O(n \times \log_2 n)$ dacă se alege un algoritm de sortare eficient, cum este de exemplu algoritmul **QuickSort**. Complexitatea metodei greedy este $O(n) + O(n \times \log_2 n) = O(n \times \log_2 n)$.

Metoda greedy se recomandă în următoarele cazuri:

- se dorește numai obținerea soluției optime și suntem siguri că aplicând strategia greedy se obține soluția optimă;
- se dorește obținerea unei soluții acceptabile, nu neapărat optime, și algoritmul greedy este mult mai eficient decât alți algoritmi – care pot duce la obținerea mai multor soluții sau a soluției optime.

 1. Un automobilist pornește dintr-un oraș A și trebuie să ajungă în orașul B. Rezervorul autoturismului, dacă este plin, îi permite să parcurgă n kilometri. Pe hartă sunt trecute m stații de alimentare și distanțele dintre ele. Automobilistul dorește să opreasca la cât mai puține stații pentru alimentarea autoturismului. Scrieți un program care să determine stațiile de benzină la care trebuie să se opreasca pentru alimentare.

2. Într-un camping există k căsuțe care pot fi închiriate 365 de zile din an. Orice turist poate închiria o căsuță pentru exact m zile consecutive din an. Campingul a adunat solicitările a n turiști pe un an întreg, cu un an înainte. Fiecare turist precizează prima zi cu care dorește să închirieze o căsuță (1, 2, ...). Să se determine numărul maxim de turiști care vor putea fi primiți în camping ($k \leq 100$, $m \leq 20$, $n \leq 1000$).

(Concursul Nemes Tihamér, 1998)

3. Un turist trebuie să străbată un traseu de munte. Pe hartă, zona prin care trebuie să treacă are o formă dreptunghiulară, care este împărțită, prin caroiaje, în subzone, care au fiecare o anumită înălțime. Zona prin care trece turistul poate fi reprezentată printr-o matrice ale cărei elemente memorează înălțimea fiecărei subzone. Turistul pornește din sud, din orice subzonă de pe hartă, și trebuie să

Nord			
2	1	1	3
2	1	2	1
0	0	4	2
3	1	2	3

Sud

ajungă în nord, în orice subzonă de pe hartă, astfel încât să urce cât mai puțin pe întreg traseul. El poate trece dintr-o zonă în alta, numai mergând în direcțile: Est, Nord-Est, Nord, Nord-Vest și Vest. Datele de intrare sunt dimensiunile matricei și matricea înălțimilor – și se vor citi din fișier.

- Într-o fabrică s-a automatizat procesul de producție și, la sfârșitul zilei de muncă, piesele sunt adunate de roboți industriali. Fiecare robot poate strânge un anumit număr de piese. Harta halei a fost împărțită printr-un caroaj și poate fi reprezentată printr-o matrice ($n \times m$) ale cărei elemente pot avea valorile: 0 (pentru locurile în care nu există mașini) și 1 (pentru locurile în care există mașini). Roboți se deplasează în spațiul de deasupra mașinilor, corespunzător acelaiași caroaj. Roboți pornesc din colțul din stânga sus (1,1) și trebuie să ducă piesele în colțul din dreapta jos (n,m). Roboți sunt programati astfel încât să se deplaseze doar la dreapta sau în jos. Să se scrie un program care să determine numărul minim de roboți care trebuie porniți pentru a strânge toate piesele fabricate într-o zi.

(Concursul Nemes Tihamér, 1998)

- Într-un depozit al monetăriei statului sosesc n saci cu monede. Șeful depozitului cunoaște numărul de monede din fiecare sac și vrea să modifice conținutul sacilor, prin mutarea de monede dintr-un sac în altul, astfel încât, la sfârșit, să fie în fiecare sac același număr de monede. Să se scrie un program care să determine, dacă este posibil, numărul minim de mutări prin care fiecare sac să contină același număr de monede, altfel să se afișeze un mesaj. Fiecare mutare se afișează sub forma: *sac inițial, sac final, număr de monede.* ($n \leq 2000$, iar numărul total de monede $\leq 1.000.000.000$)

(ONI Oradea, 1998)

1.6. Metoda programării dinamice

1.6.1. Descrierea metodei programării dinamice

Metoda programării dinamice se poate folosi pentru problemele în care trebuie să fie găsită **soluția optimă** și care au următoarele caracteristici:

- Soluția optimă se alege dintr-o mulțime de soluții, fiecărei soluții putând să i se asocieze o **valoare**. Alegerea soluției optime înseamnă alegerea soluției care are **valoarea optimă** (minimă sau maximă).
- Problema poate fi descompusă în **subprobleme similare cu problema inițială** ce respectă **principiul optimalității**: soluția problemei este optimă dacă ea conține soluțiile optime ale subproblemelor.
- Subproblemele în care se descompune problema nu sunt independente.
- Soluția** problemei este dată de un **vector $S = \{x_1, x_2, \dots, x_m\}$** și:
 - există o mulțime finită A din care se poate alege un element x_i al soluției;
 - fiecare etapă de determinare a unui element x_i al soluției se bazează pe rezultatele etapelor în care s-au determinat elementele anterioare ale soluției;
 - numărul de posibilități de a alege un element x_i al soluției se reduce din cauza cerințelor de optimizare și a restricțiilor impuse soluției.

Metoda clasică de rezolvare a acestor probleme este de a căuta toate soluțiile problemei și de a alege, dintre soluțiile găsite, soluția optimă. Algoritmul de generare a tuturor soluțiilor înseamnă generarea tuturor submulțimilor de indici ai mulțimii A cu n elemente și are **ordinul de complexitate $O(2^n)$** . Acest ordin de complexitate este inacceptabil pentru o dimensiune mare a datelor de intrare n .

Studiu de caz

Scop: identificarea problemelor care pot fi descompuse în subprobleme similare care nu sunt independente și care respectă principiul optimalității, soluțiilor putând să li se asocieze o valoare, iar soluția optimă determinându-se prin calcularea valorii optime.

Enunțul problemei 1: Se consideră un triunghi de numere naturale a_{ij} cu n linii. Pornind de la numărul din linia 1, mergând în jos până la linia n , să se determine o selecție de elemente astfel încât **suma elementelor să fie maximă**. Trecerea la linia următoare se poate face numai mergând în jos, direct sau pe diagonală, la dreapta.

a_{11}				
a_{21}	a_{22}			
.....				
a_{n1}	a_{n2}	...	a_{nn}	

Metoda clasică de rezolvare a acestei probleme este de a calcula toate sumele care se pot forma respectând restricția de deplasare în triunghi și de a alege, dintre sumele găsite, suma care are valoarea cea mai mare.

Mulțimea datelor de intrare o reprezintă mulțimea A formată din elementele a_{ij} ale triunghiului. **Valoarea** asociată unei soluții este **suma elementelor**. Valoarea soluției optime este valoarea maximă (**suma elementelor trebuie să fie maximă**). **Soluția** problemei este dată de un vector $S = \{x_1, x_2, \dots, x_n\}$, unde $x_i \in A$. Pentru construirea soluției se pornește de la elementul a_{11} și se merge în jos, până la linia n . Trecerea la linia următoare nu se poate face decât direct, în jos sau pe diagonală, la dreapta, astfel încât successorul elementului a_{ij} nu poate fi decât elementul $a_{i+1,j}$ sau elementul $a_{i+1,j+1}$. Problema poate fi descompusă în subprobleme: subproblema i este alegerea elementului x_i al soluției de pe linia i a triunghiului. Restricția impusă pentru soluție este ca elementul x_i al soluției să se obțină, prin deplasarea în triunghi, numai pe două direcții, de la elementul anterior, limitând mulțimea elementelor din care se alege elementul x_i al soluției numai la elementele din triunghi care se găsesc într-o anumită poziție față de elementul anterior al soluției. Pentru a respecta **principiul optimalității**, suma elementelor alese anterior trebuie să fie maximă. Subproblemele nu sunt independente deoarece în subproblema i alegerea elementului x_i al soluției se bazează pe alegerile făcute în subproblemele anterioare.

Enunțul problemei 2: Dându-se un sir de numere să se determine **subșirul crescător de lungime maximă**.

Fiind dat un sir de numere a_1, a_2, \dots, a_n , se definește ca subșir crescător, sirul $a_{i1}, a_{i2}, \dots, a_{ik}$, cu $a_{i1} \leq a_{i2} \leq \dots \leq a_{ik}$ și $1 \leq i_1 < i_2 < \dots < i_k$. De exemplu, în sirul de numere 1, -2, 3, 2, 4, 4, un subșir crescător este 1, 3, 4, 4.

Metoda clasică de rezolvare a acestei probleme este de a genera fiecare subșir al mulțimii A și de a verifica dacă este subșir crescător. Dintre subșirurile crescătoare găsite se alege apoi subșirul care are lungimea cea mai mare.

Mulțimea datelor de intrare o reprezintă mulțimea formată din elementele sirului: $A = \{a_1, a_2, \dots, a_n\}$. **Valoarea** asociată unei soluții este **lungimea subșirului crescător**. Valoarea soluției optime este valoarea maximă (**lungimea subșirului trebuie să fie maximă**). **Soluția** problemei este dată de un vector $S = \{x_1, x_2, \dots, x_m\}$, unde $x_i \in A$. Problema poate fi descompusă în subprobleme: subproblema i este de a găsi subșirul crescător de lungime maximă care începe cu elementul a_i . Restricția impusă pentru soluție este ca elementul a_i cu care începe subșirul să aibă valoarea mai mică sau cel mult egală cu cea a successorului său în subșir (a_j), iar în mulțimea A numărul lui ordine (i) trebuie să fie mai mic decât numărul de ordine al successorului (j), limitând mulțimea elementelor din care se alege succesorul a_j . Metoda cea mai simplă de a găsi subșirul de lungime maximă care începe cu

elementul a_i este de a căuta printre subşirurile de lungime maximă descoperite în subproblemele rezolvate anterior, cel mai lung subşir care începe cu un element a_j ce poate fi succesor în subşir al elementului a_i , obținându-se un nou subşir, de lungime mai mare, care începe cu elementul a_j . Pentru a respecta principiul optimalității, lungimea subşirurilor descoperite anterior trebuie să fie maximă. Subproblemele nu sunt independente deoarece în subproblema i alegerea subşirului care începe cu elementul a_j al soluției se bazează pe alegerile făcute în subproblemele anterioare.

Enunțul problemei 3: O țeavă cu lungimea L trebuie să se confectioneze din n bucăți de țeavă, fiecare bucată având lungimea a_i . O bucată de țeavă nu poate fi tăiată. Să se găsească, dacă există, soluția de a obține țeava de lungime L prin sudarea unor bucăți de țeavă cu lungimea a_i .

Metoda clasică de rezolvare a acestei probleme este de a genera toate țevile care se pot forma cu cele n bucăți de țeavă cu lungimile date și de a verifica dacă lungimea țevii obținute este egală cu L .

Mulțimea datelor de intrare o reprezintă mulțimea A formată din lungimile bucăților de țeavă a_i . Valoarea asociată unei soluții este lungimea țevii. Valoarea soluției optime este L (lungimea țevii construite trebuie să fie L). Soluția problemei este dată de un vector $S = \{x_1, x_2, \dots, x_m\}$, unde $x_i \in A$. Problema poate fi descompusă în subprobleme: subproblema i este de a găsi toate țevile care se pot confectiona adăugând bucată de țeavă cu lungimea a_i la țevile care au fost construite în subproblema anterioară. Restricția impusă pentru țevile care se construiesc este ca ele să nu depășească lungimea L . Această restricție limitează mulțimea țevilor construite anterior. Pentru a respecta principiul optimalității, construcția țevilor cu ajutorul bucății de țeavă cu lungimea a_i se bazează numai pe țevile construite în subproblema anterioară, iar subproblemele nu sunt independente.

Enunțul problemei 4: Într-un rucsac se poate transporta o greutate maximă G și există n obiecte, fiecare obiect având greutatea g_i și un profit obținut în urma transportului c_i , iar obiectele nu pot fi fracționate. Să se selecteze obiectele care vor fi transportate în rucsac, astfel încât să se obțină profitul maxim (problema **discretă a rucsacului**).

Metoda clasică de rezolvare a acestei probleme este de a genera toate posibilitățile de a încărca rucsacul alegând obiecte din cele n obiecte și de a verifica dacă greutatea obiectelor nu depășește greutatea de încărcare a rucsacului G . Dintre variantele de încărcare a rucsacului se alege cea care are profitul cel mai mare.

Mulțimea datelor de intrare o reprezintă mulțimea A formată din cele n obiecte a_i care pot fi transportate în rucsac. Valoarea asociată unei soluții este profitul transportului. Valoarea soluției optime este valoarea maximă (profitul trebuie să fie maxim). Soluția problemei reprezintă încărcătura optimă a rucsacului și este dată de un vector $S = \{x_1, x_2, \dots, x_m\}$, unde $x_k \in A$ și $1 \leq k \leq m$. Problema poate fi descompusă în subprobleme: subproblema i este de a verifica dacă obiectul a_i poate fi element al soluției. Restricția impusă pentru soluția care se construiește este ca obiectul a_i să nu aibă o greutate mai mare decât greutatea disponibilă a rucsacului și profitul lui să fie mai mare sau egal cu profitul ultimului obiect adăugat în rucsac a_{i-1} . Această restricție limitează mulțimea elementelor din care se alege elementul x_k al soluției. Pentru a respecta principiul optimalității, elementul x_k al soluției se determină prin alegerea dintre două obiecte a celui care are profitul mai mare.

Subproblemele nu sunt independente deoarece rezolvarea subproblemei i se bazează pe rezolvarea subproblemei $i-1$ (elementul x_i al soluției este determinat în funcție de elementul x_{i-1} al soluției).



Metoda **programării dinamice** se bazează pe descompunerea unei probleme în subprobleme **similar problemei inițiale**, care **nu sunt independente**, și construiește soluția rezolvând fiecare subproblemă, prin selectarea, dintr-o mulțime de elemente, a elementelor care îndeplinesc o anumită condiție. Pentru ca elementele care se selectează să aparțină **soluției optime**, la pasul k alegera se face în funcție de valoarea optimă a unei **funcții asociate** soluției și se bazează pe alegerile care s-au făcut până atunci (pe elementele soluției care au fost descoperite anterior).

Etapele de rezolvare a unei probleme prin metoda programării dinamice sunt:

- PAS1.** Se demonstrează că problema are o substructură optimă (prin reducere la absurd).
- PAS2.** Se caracterizează structura unei soluții optime (descompunerea problemei în subprobleme).
- PAS3.** Se definește **recursiv** valoarea soluției optime.
- PAS4.** Se calculează valoarea soluției optime într-o manieră „**de jos în sus**” („bottom-up”).
- PAS5.** Se construiește soluția optimă din informațiile obținute prin calcularea valorii soluției optime.

Observație. Valoarea soluției optime este definită recursiv în funcție de valorile optime ale subproblemelor. Dacă pentru calcularea funcției recursive s-ar folosi un algoritm recursiv, ordinul de complexitate ar fi $O(2^n)$, la fel ca și în cazul metodei clasice de rezolvare. Din această cauză, calcularea funcției recursive se face „**de jos în sus**” obținându-se un **algoritm iterativ**:

- PAS1.** Se rezolvă subproblema obținută din descompunerea problemei inițiale care are cea mai mică dimensiune și se memorează soluția ei.
- PAS2.** Cât timp nu s-a ajuns la rezolvarea problemei cu dimensiunea **n**, execută: se rezolvă o subproblemă de dimensiunea **i** folosind soluțiile obținute în subproblemele cu dimensiune mai mică decât **i**.

1.6.2. Implementarea metodei programării dinamice

Pentru implementarea metodei programării dinamice se vor folosi subprogramele:

- subprogramul **init()** – care initializează variabilele de memorie și structurile de date folosite pentru construirea soluției;
- subprogramul **p_dinamica()** – care calculează valoarea soluției optime;
- subprogramul **afiseaza()** – care afișează soluția problemei folosind informațiile obținute prin calcularea valorii soluției optime.

În exemplele următoare datele de intrare vor fi citite dintr-un fișier text.

Exemplul 1. Problema **sumei maxime în triunghi**.

Cerința problemei este ca pentru calculul sumei să se pornească de la elementul **a₁₁** și să se meargă în jos, până la linia **n**. Trecerea la linia următoare nu se poate face decât direct, în jos, sau pe diagonală, la dreapta, astfel încât succesorul elementului **a_{ij}** nu poate fi decât elementul **a_{i+1,j}** sau elementul **a_{i+1,j+1}**.

PAS1. Problema are substructură optimă. Fie un șir de **n** numere din triunghi, care respectă condițiile de deplasare în triunghi ale problemei, și care formează suma maximă. În subproblema **i** a fost ales numărul de pe linia **i** pe baza sumelor maxime calculate în subproblemele anterioare, altfel se contrazice ipoteza că șirul formează suma maximă.

PAS2. Descompunerea problemei în subprobleme. Se determină pe rând sumele maxime ale numerelor care apar pe traseele care pornesc de pe ultima linie n către vârf și ajung pe linia i ($1 \leq i \leq n-1$). Pe linia i , suma maximă pentru fiecare element se va calcula pe baza sumelor maxime care s-au calculat în subproblemele anterioare (sumele de la linia n până la linia $i+1$). Aceste sume sunt sumele maxime pentru fiecare linie și principiul optimalității este respectat.

PAS3. Valoarea soluției fiind suma elementelor selectate, funcția recursivă care definește **suma maximă** pornind de la elementul a_{ij} al triunghiului este prezentată alăturat.

$$S(i,j) = \begin{cases} a_{ij} & \text{dacă } i=n, 1 \leq j \leq 1 \\ a_{ij} + \max(S(i+1,j), S(i+1,j+1)) & \text{dacă } 1 \leq i \leq n-1 \end{cases}$$

PAS5. Pentru a calcula valoarea soluției optime într-o manieră de „de jos în sus”, se va forma un **triunghi al sumelor maxime**, pornind de la bază (linia n) către vârf (linia 1). Sumele maxime de pe linia n sunt egale fiecare cu elementul corespunzător de pe linia n a triunghiului. Valoarea maximă a sumei elementelor este S_{11} .

PAS5. Pentru reconstituirea elementelor soluției, se folosește o matrice p , în care se memorează direcția de deplasare (valoarea 1 pentru deplasarea în jos și valoarea 2 pentru deplasarea pe diagonală). Matricea p este definită alăturat.

$$p(i,j) = \begin{cases} 1 & \text{dacă } S(i+1,j) > S(i+1,j+1) \\ 2 & \text{dacă } S(i+1,j) \leq S(i+1,j+1) \end{cases}$$

Se folosesc următoarele date și structuri de date:

→ Pentru numărul de liniile ale triunghiului se folosește variabila n .

→ Pentru memorarea triunghiului se folosește matricea a , pentru memorarea sumelor maxime – matricea S , iar pentru memorarea direcției de deplasare – matricea p . Toate aceste matrice sunt pătrate, cu dimensiunea n .

Se folosesc următoarele subprograme:

$$\begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \dots & \dots & \dots & 0 \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

→ Subprogramul `citeste()` – pentru citirea datelor din fișierul text. În fișierul text, pe primul rând este scris numărul de liniile, n , iar pe următoarele n rânduri numerele de pe fiecare linie a triunghiului. Numerele sunt separate prin spațiu. În urma execuției acestui subprogram se creează matricea triunghiului a .

→ Subprogramul `init()` – inițializează linia n a matricei S cu elementele de pe linia n a triunghiului.

→ Subprogramul `p_dinamica` – calculează valoarea soluției optime (elementul S_{11} al matricei sumelor).

→ Subprogramul `afiseaza()` – afișează suma maximă și soluția problemei folosind informațiile din matricea p .

Strategia programării dinamice este implementată astfel:

PAS1. Se inițializează linia n a matricei sumelor maxime S cu elementele de pe linia n a matricei triunghiului a .

PAS2. Pentru următoarele liniile ale matricei sumelor maxime începând cu linia $n-1$ până la linia 1, **execută**:

PAS3. Se calculează suma maximă astfel: dacă suma maximă $S(i+1,j)$ – de sub elementul curent din triunghi – este mai mare decât suma maximă $S(i+1,j+1)$ – de pe diagonala elementului curent, atunci la elementul

current se adună suma $S(i+1,j)$ și direcția de deplasare este 1; altfel, la elementul current se adună suma $S(i+1,j+1)$ și direcția de deplasare este 2.

Matricea triunghiului - a

Suma maximă = 20

	1	2	3	4
1	5	0	0	0
2	4	2	0	0
3	5	4	3	0
4	4	6	2	5

Matricea sumelor

maxime - S

	1	2	3	4
1	20	0	0	0
2	15	1	0	0
3	11	10	8	0
4	4	6	2	5

Matricea direcției de

deplasare - p

	1	2	3	4
1	1	0	0	0
2	1	1	0	0
3	2	1	2	0
4	0	0	0	0

Programul este:

```
#include<iostream.h>
int n,a[20][20],p[20][20],S[20][20];
fstream f("pdl.txt",ios::in);
void citeste()
{int i,j; f>>n;
for(i=1;i<=n;i++)
    for(j=1;j<=i;j++) f>>a[i][j]; f.close(); }
void init()
{for (int i=1;i<=n;i++) S[n][i]=a[n][i];}
void p_dinamica()
{int i,j;
for (i=n-1;i>=1;i--)
    for (j=1;j<=i;j++)
        if (S[i+1][j]>S[i+1][j+1])
            {S[i][j]=a[i][j]+S[i+1][j]; p[i][j]=1;}
        else {S[i][j]=a[i][j]+S[i+1][j+1]; p[i][j]=2; } }
void afiseaza()
{int i,j,q=1,r=1;
cout<<"Suma maxima= "<<S[1][1]<< "=" <<a[1][1]<<"+";
for (i=1,j=1;i<n;i++)
    {if (p[i][j]==2) {r=r+1; j++;}
     q=q+1; cout<<a[q][r]<<"+";}
cout<<'b'<< " <<endl; }
void main()
{citeste(); init(); p_dinamica(); afiseaza(); }
```

Exemplul 2. Problema șirului de lungime maximă.

Pentru a determina șirul de lungimea maximă, trebuie calculată, pentru fiecare element a_i al șirului, lungimea celui mai mare subșir crescător care se poate forma începând cu el, după care se alege subșirul cu lungimea maximă.

PAS1. Problema are substructură optimă. Fie un șir crescător de m numere din șirul inițial A care începe cu numărul a_i și respectă condiția că este subșir al șirului A, și care are lungimea maximă dintre toate subșirurile crescătoare care începe cu numărul a_i . În subproblemă i se alege subșirul crescător de lungime maximă care începe cu numărul a_i astfel: se adaugă la numărul a_i subșirul cu lungime maximă dintre subșirurile din subproblemele ante-

rioare care încep cu numărul a_j , cu $a_i \leq a_j$ și $i < j$, altfel se contrazice ipoteza că subșirul are lungimea maximă.

PAS2. Descompunerea problemei în subprobleme. Se determină pe rând lungimea maximă a unui subșir crescător care începe cu elementul a_i din sir, pornind de la subșirurile crescătoare de lungime maximă care încep cu elementul a_n , până la subșirurile care încep cu elementul a_{i+1} . Pentru elementul a_i din sir, lungimea maximă a subșirurilor crescătoare care se pot forma, începând cu el, se va calcula pe baza lungimilor maxime care s-au calculat în subproblemele anterioare. Aceste lungimi sunt lungimile maxime pentru fiecare subșir care se poate forma începând cu acel element și principiul optimalității este respectat.

PAS3. Valoarea soluției fiind lungimea subșirului crescător, funcția recursivă care definește **lungimea maximă a subșirului** care se poate forma pornind de la elementul a_i al sirului este prezentată alăturat.

$$L(i) = \begin{cases} 1 & \text{dacă } i=n \\ 1 + \max(L(j)) & \text{dacă } a_i \leq a_j, i \neq n \text{ și } i < j \leq n \end{cases}$$

PAS4. Pentru a calcula valoarea soluției optime într-o manieră „de jos în sus”, se va forma un vector **al lungimilor maxime**, pornind de la ultimul element din sir (elementul n) către primul element din sir. În acest vector se va memora în poziția i lungimea maximă a unui subșir crescător care se poate forma începând cu elementul a_i al sirului. Cum lungimea maximă a unui subșir crescător care se poate forma cu un singur element este 1, lungimea maximă corespunzătoare elementului a_n se inițializează cu valoarea 1.

PAS5. Pentru reconstituirea elementelor soluției, se folosește vectorul p în care, în poziția i , se memorează indicele elementului următor din subșirul de lungime maximă care începe cu a_i .

Se folosesc următoarele date și structuri de date:

- Pentru numărul de elemente ale sirului se folosește variabila n .
- Pentru memorarea sirului de numere se folosește vectorul a , pentru memorarea lungimilor maxime – vectorul L , iar pentru memorarea indicelui elementului următor din subșir – vectorul p . Toți acești vectori au dimensiunea n .

Se folosesc următoarele subprograme:

- Subprogramul citeste() – pentru citirea datelor din fișierul text. În fișierul text, pe primul rând este scrisă lungimea sirului – n , iar pe următorul rând cele n numere din sir. În urma execuției acestui subprogram se creează vectorul pentru sirul de numere – a .
- Subprogramul init() – inițializează elementul n al vectorilor L și p .
- Subprogramul p_dinamica – calculează valoarea soluției optime (lungimea maximă a subșirurilor care încep cu fiecare element a_i din sir).
- Subprogramul cauta_soluția() – caută cea mai mare lungime maximă a unui subșir pentru a găsi elementul cu care începe subșirul.
- Subprogramul afiseaza() – afișează lungimea sirului găsit și soluția problemei folosind informațiile din vectorul p .

Strategia programării dinamice este implementată astfel:

PAS1. Se inițializează elementul n al vectorului lungimilor maxime cu 1 și al vectorului p cu n .

PAS2. Pentru următoarele elemente i ale vectorului, lungimilor maxime începând cu elementul $n-1$ până la elementul 1, execută:

PAS3. Se inițializează lungimea maximă a subșirului care începe cu acest

Indici	1	2	3	4	5	6
a	1	-2	3	2	4	4
L	4	4	3	3	2	1
p	3	3	5	5	6	6
Subșirul	1		3		4	4

element cu valoarea 1 și poziția cu care începe subșirul cu poziția elementului în sir.

PAS4. Se calculează lungimea maximă a subșirului care începe cu elementul curent din sir, astfel: pentru toate elementele j din sir care urmează după elementul i , execută

PAS5. Dacă a_i este mai mic decât a_j și lungimea subșirului care începe cu a_j este mai mare decât lungimea subșirului care începe cu a_i , atunci $L(i)$ este egal cu $1+L(j)$ și indicele elementului care urmează în subșir după elementul a_i este j .

PAS6. Se determină cea mai mare lungime maximă $L(k)$. Subșirul de lungime maximă va începe cu elementul a_k .

Programul este:

```
#include<iostream.h>
int n,k,a[20],p[20],L[20];
ifstream f("pd2.txt",ios::in);
void citeste()
{int i; f>>n;
 for(i=1;i<=n;i++) f>>a[i]; f.close();}
void init()
{L[n]=1; p[n]=n;}
void p_dinamica()
{int i,j;
 for (i=n-1;i>=1;i--)
 {L[i]=1; p[i]=i;
 for (j=i+1;j<=n;j++)
 if (a[i]<=a[j] && L[i]<=L[j])
 {L[i]=L[j]+1; p[i]=j;}}
void cauta_solutie()
{int i,max=-1;
 for (i=1;i<=n;i++)
 if (max<L[i]) {max=L[i]; k=i;}}
void afiseaza()
{int i,j;
 cout<<"Lungime subșir= "<<L[k]<<endl;
 for (i=1,j=k;i<=L[k];i++,j=p[j]) cout<<a[j]<<" ";}
void main()
{citeste();p_dinamica(); cauta_solutie(); afiseaza(); }
```

Exemplul 3. Problema țevii.

Dacă se poate construi o țeavă de lungimea D folosind bucăți de țeavă cu lungimea a_1, a_2, \dots, a_i , atunci se poate construi și o țeavă cu lungimea $D+a_{i+1}$ folosind bucăți de țeavă cu lungimea $a_1, a_2, \dots, a_i, a_{i+1}$.

PAS1. Problema are substructură optimă. Fie o țeavă care s-a obținut prin sudarea bucății de țeavă cu lungimea a_i , la o țeavă care a fost obținută anterior din bucăți de țeavă cu lungimea a_j , și care are lungimea mai mică decât L . În subproblema i se aleg dintre țevile obținute anterior numai acelea care au lungimea mai mică decât $L-a_i$, altfel se contrazice ipoteza că se poate construi noua țeavă.

PAS2. Descompunerea problemei în subprobleme. Se determină pe rând lungimile ţevilor care se pot forma cu bucata de ţeavă cu lungimea a_i , pornind de la ţeava formată numai cu bucata de ţeavă cu lungimea a_1 , până la ţevile formate cu primele $i-1$ bucăți de ţeavă. Ţevile formate nu depășesc lungimea L și principiul optimalității este respectat.

PAS3. Valoarea soluției fiind lungimea unei ţevi, funcția recursivă care definește **lungimea ţevilor** care se pot forma cu bucata de ţeavă cu lungimea a_i este prezentată alăturat.

$$T(i,j) = \begin{cases} -1 & \text{dacă } j=0 \\ a_i & \text{dacă } T(i,j-a_i) \neq 0 \text{ pentru } 1 \leq i \leq n \text{ și } 1 \leq j \leq L \end{cases}$$

PAS4. Pentru a calcula valoarea soluției optime într-o manieră „de jos în sus”, se vor forma toate ţevile cu lungimea mai mică sau egală cu L , pornind de la ţeava care se poate construi cu bucata de ţeavă cu lungimea a_1 , și adăugând la ţevile obținute, pe rând, următoarele $n-1$ bucăți de ţeavă cu lungimea a_i . Pentru a ține evidența ţevilor care s-au construit în subproblema i se folosește un **vector al lungimilor ţevilor** care se construiesc – T , care are lungimea logică L . În acest vector se va memora în poziția j lungimea ultimei bucăți de ţeavă care se adaugă la una dintre ţevile construite anterior pentru a obține o ţeavă cu lungimea j . Principiul optimalității este respectat prin modul în care se construiește o ţeavă cu lungimea j : prin adăugarea unei bucăți de ţeavă la una dintre ţevile confectionate în subproblema anterioară.

PAS5. Pentru reconstituirea elementelor soluției, se folosește vectorul T . Pornind de la elementul din poziția L care corespunde ţevii cu lungimea L , se scade bucata de ţeavă cu lungimea $T(L)$, obținându-se lungimea ţevii la care a fost adăugată această bucătă de ţeavă (indicele din vectorul T). Procesul continuă până se obține ţeava cu lungimea 0.

Se folosesc următoarele date și structuri de date:

- Pentru numărul de bucăți de ţeavă se folosește variabila n , pentru lungimea ţevii care trebuie confectionată, variabila L , iar pentru lungimea maximă a unei ţevi care se poate forma adăugând o nouă bucătă de ţeavă la ţevile construite anterior – variabila \max . Variabila \max are inițial valoarea 0 (nu s-a construit nici o ţeavă), iar la sfârșit, dacă problema are soluție, are valoarea L (lungimea ţevii care trebuie confectionată).
- Pentru memorarea lungimii fiecarei bucăți de ţeavă se folosește vectorul a , de lungime n , iar pentru memorarea ţevilor care se pot confectiona, vectorul T , de lungime L .

Se folosesc următoarele subprograme:

- Subprogramul **citestे()** – pentru citirea datelor din fișierul text. În fișierul text, pe primul rând sunt scrise numărul de bucăți de ţeavă – n și lungimea ţevii – L , iar pe următorul rând, lungimile celor n bucăți de ţeavă. În urma execuției acestui subprogram se creează vectorul cu lungimile bucățiilor de ţeavă – a .
- Subprogramul **init()** – inițializează elementul 0 al vectorului T și variabila \max .
- Subprogramul **p_dinamica** – calculează valoarea soluției optime (lungimile ţevilor ce se pot obține adăugând fiecare bucătă de ţeavă a_i).
- Subprogramul **soluție()** – verifică dacă s-a găsit soluția problemei.
- Subprogramul **afiseaza()** – dacă este posibil să se confectioneze ţeava, afișează lungimile bucățiilor de ţeavă din care se confectionează.

Strategia **programării dinamice** este implementată astfel:

- PAS1.** Se inițializează elementul 0 al vectorului T cu -1 și variabila \max cu valoarea 0.
- PAS2.** Pentru fiecare bucătă de ţeavă cu lungimea a_i începând cu bucata de ţeava cu lungimea a_1 până la bucata de ţeavă cu lungimea a_n , execută:

PAS3. Pentru țevile cu lungimea j începând cu țeava cu lungimea **max** până la țeava cu lungimea 1, **execută**:

PAS4. Dacă adăugând la țeava cu lungimea j bucata de țeavă cu lungimea a_i se obține o țeavă care va putea fi folosită la confectionarea țevii cu lungimea L , atunci construirea țevii cu lungimea $j+a_i$ începe cu bucata de țeavă cu lungimea a_i . Se revine la Pasul 3.

PAS5. Se determină lungimea maximă a unei țevi care se poate construi din țevile construite anterior, astfel: dacă adăugând la țeava cu lungimea maximă bucata de țeavă cu lungimea a_i , se obține o țeavă cu lungimea mai mică decât L , atunci lungimea maximă a unei țevi care se poate construi va fi mărită cu lungimea bucătii de țeavă a_i ; altfel, ea va fi L . Se revine la Pasul 2.

PAS6. Dacă nu există soluție (nu există o bucătă de țeavă cu care se poate începe construirea țevii de lungime L), atunci se afișează un mesaj de informare; altfel, se trece la Pasul 7 pentru a afișa soluția.

PAS7. Pentru fiecare țeavă cu lungimea i din care s-a obținut țeava finală, începând de la țeava cu lungimea L , până la țeava cu lungimea 0, **execută**:

PAS8. Se afișează lungimea bucătii de țeavă cu care începe construirea țevii i : $T(i)$.

PAS9. Se calculează lungimea țevii rămase, scăzând lungimea bucătii de țeavă din lungimea i .

$L=10$	indici	1	2	3	4	5
$n=5$	a	2	2	3	5	6

indici vectorul $T = \text{lungime } \tau_{\text{eavă}}$

max	a(i)	0	1	2	3	4	5	6	7	8	9	10
initial	-1	0	0	0	0	0	0	0	0	0	0	0
0	a(1)	-1	0	2	0	0	0	0	0	0	0	0
2	a(2)	-1	0	2	0	2	0	0	0	0	0	0
4	a(3)	-1	0	2	3	2	3	0	3	0	0	0
7	a(4)	-1	0	2	3	2	5	0	5	5	5	5
10	a(5)	-1	0	2	3	2	5	6	5	6	6	6

De exemplu, după ce s-a luat în considerație și bucata de țeavă de lungime $a(2)=2$ se pot construi țevi cu lungimea 2 (din bucata de țeavă $a(1)=2$) și cu lungimea 4 (din bucătările de țeavă $a(1)=2$ și $a(2)=2$), lungimea maximă a unei țevi fiind 4. Luându-se în considerație și bucata de țeavă de lungime $a(3)=3$ se pot construi țevi cu lungimea 2 (din bucata de țeavă $a(1)=2$), cu lungimea 3 (din bucata de țeavă $a(3)=3$), cu lungimea 4 (din bucătările de țeavă $a(1)=2$ și $a(2)=2$) și cu lungimea 5 (din bucătările de țeavă $a(1)=2$ și $a(3)=3$).

Programul este:

```
#include<fstream.h>
int L,n,max,T[20],a[20];
fstream f("pd3.txt",ios::in);
void citeste()
{int i; f>>L>>n;
 for(i=1;i<=n;i++) f>>a[i]; f.close();}
void init()
{T[0]=-1; max=0;}
```

```

void p_dinamica()
{int i,j,k;
for (i=1;i<=n;i++)
{for (j=max;j>=0;j--)
    if (T[j]!=0 && j+a[i]<=L) T[j+a[i]]=a[i];
    if (max+a[i]<L) max=max+a[i]; else max=L;}
int solutie()
{return T[L]!=0;}
void afiseaza()
{if (!solutie()) cout<<"Teava nu se poate confectiona";
else for (int i=L;i!=0;)
    {cout<<"Bucata de "<<T[i]<<" metri "<<endl;
     i-=T[i];}}
void main()
{citere(); init(); p_dinamica(); afiseaza();}

```

Exemplul 4. Problema discretă a rucsacului.

PAS1. Fie un sir de m obiecte a căror greutate nu depășește greutatea de încărcare a rucsacului G și care asigură profitul maxim al transportului. În subproblemă i a fost ales obiectul i numai dacă greutatea sa nu depășește greutatea de încărcare disponibilă și profitul adus de el este mai mare decât al obiectului adăugat anterior în rucsac, altfel se contrazice ipoteza că încărcarea rucsacului asigură profitul maxim.

PAS2. Descompunerea problemei în subprobleme. Se determină pe rând profitul maxim adus de fiecare obiect a_i pentru fiecare greutate de încărcare disponibilă (de la 1 la G) pornind de la obiectul a_1 , până la obiectul a_n , pe baza profiturilor maxime aduse de primele $i-1$ obiecte.

PAS3. Valoarea soluției fiind profitul, funcția recursivă care definește **profitul maxim $C(i,j)$** obținut prin încărcarea optimă a rucsacului cu primele i obiecte, având disponibilă greutatea de încărcare a rucsacului j , este prezentată alăturat. $C(i,0)$ reprezintă profitul maxim al unui rucsac cu greutatea disponibilă 0 (care nu permite adăugarea niciunui obiect i). $C(0,j)$ reprezintă profitul maxim al unui rucsac gol cu greutatea disponibilă j (care nu conține niciun obiect).

PAS3. Pentru a calcula valoarea soluției

optime într-o manieră „de jos în sus”, se va forma o **matrice a profiturilor maxime C** , pornind de la primul obiect (linia 1), până la ultimul obiect (linia n). Matricea C are dimensiunea $n \times G$. Elementul $C(i,j)$ memorează profitul maxim obținut în subproblemă i pentru o greutate disponibilă j . Pe linia i , profitul maxim pentru fiecare greutate de încărcare j se determină pe baza profiturilor maxime ce s-au calculat în subproblemele anterioare (profiturile maxime de la linia 1 până la linia $i-1$). Aceste profituri sunt maxime pentru fiecare linie și principiul optimalității este respectat. Valoarea maximă a profitului este C_{nG} (profitul maxim obținut pentru un rucsac cu greutatea disponibilă G ocupat prin alegerea din cele n obiecte a obiectelor care aduc profitul maxim).

PAS4. Pentru reconstituirea elementelor soluției, se folosește matricea p cu dimensiunea $n \times G$. Elementul $p(i,j)$ memorează obiectul ales în subproblemă i pentru o greutate disponibilă j .

$$C(i,j) = \begin{cases} 0 & \text{dacă } j=0 \text{ și } 1 \leq i \leq n \\ 0 & \text{dacă } i=0 \text{ și } 1 \leq j \leq G \\ \max(C(i-1,j-g(i))+c(i), C(i-1,j)) & \text{dacă } g(i) \leq j \\ C(i-1,j) & \text{în rest} \end{cases}$$

Se folosesc următoarele date și structuri de date globale:

- Pentru numărul de obiecte se folosește variabila n , iar pentru greutatea rucsacului, variabila G .
- Pentru multimea obiectelor (A) se folosește vectorul a ale cărui elemente sunt de tip înregistrare obiect care conține două câmpuri: g – greutatea obiectului și c – profitul obținut în urma transportului.
- Pentru memorarea profiturilor se folosește matricea C , iar pentru memorarea obiectelor alese într-o subproblemă se folosește matricea p . Ambele matrice au dimensiunea $n \times G$.

Se folosesc următoarele subprograme:

- Subprogramul `citeste()` – pentru citirea datelor din fișierul text. În fișierul text, pe primul rând este scris numărul de obiecte n și greutatea maximă a transportului G , iar pe următoarele n rânduri perechi de numere întregi, separate prin spațiu, care reprezintă greutatea și profitul transportului unui obiect. În urma execuției acestui subprogram se creează vectorul obiectelor a .
- Subprogramul `p_dinamica` – calculează valoarea soluției optime (profitul maxim obținut pentru un rucsac cu greutatea disponibilă G ocupat prin alegerea din cele n obiecte, a obiectelor care aduc profitul maxim).
- Subprogramul `afiseaza()` – afișează profitul maxim și soluția problemei folosind informațiile din matricea p .

$n=4 \ G=10$

Obiectul	Greutatea	Profitul
1	2	20
2	3	40
3	6	50
4	4	45

Matricea C

Obiect	Greutatea disponibilă									
	1	2	3	4	5	6	7	8	9	10
1	0	20	20	20	20	20	20	20	20	20
2	0	20	40	40	60	60	60	60	60	60
3	0	20	40	60	60	60	60	70	90	90
4	0	20	40	45	60	65	85	85	105	105

Matricea p

Greutatea disponibilă

Obiect	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	1	1	1	1	1	1
2	0	1	2	2	2	2	2	2	2	2
3	0	1	2	2	2	2	2	3	3	3
4	0	2	4	2	4	4	4	4	4	4

Soluția

Obiectul	Greutatea	Profitul
4	4	45
2	3	40
1	2	20
Total	9	115

Strategia programării dinamice este implementată astfel:

PAS1. Pentru fiecare obiect începând cu primul obiect până la ultimul obiect (n), execută:

 | **PAS2.** Pentru greutatea disponibilă j începând de la 1 până la G , execută:

 | **PAS3.** Se calculează profitul maxim pentru greutatea disponibilă j luând în considerație profitul care s-ar putea obține prin adăugarea obiectului i : dacă greutatea obiectului i este mai mică sau egală cu greutatea disponibilă și profitul adus la greutatea j este mai mare decât profitul adus de obiectul $i-1$, atunci profitul pentru obiectul i și greutatea j este egal cu profitul obiectului i adunat la profitul calculat pentru obiectul $i-1$ și greutatea j diminuată cu greutatea obiectului i și în

matricea p se memorează obiectul i; altfel, profitul pentru obiectul i și greutatea j este egal cu profitul obiectului i-1 și greutatea j și în matricea p se memorează obiectul i-1 ales pentru greutatea j.

Programul este:

```
#include<iostream.h>
struct obiect {int g,c;};
obiect a[20];
int n,G,p[20][20],C[20][20];
fstream f ("pd4.txt",ios::in);
void citeste()
{int i; f>>n>>G;
 for(i=1;i<=n;i++) f>>a[i].g>>a[i].c; f.close();}
void p_dinamica()
{for (int i=1;i<=n;i++)
    for (int j=1;j<=G;j++)
        if ((a[i].g<=j) && (a[i].c+C[i-1][j-a[i].g]>C[i-1][j]))
            {C[i][j]=a[i].c+C[i-1][j-a[i].g]; p[i][j]=i;}
        else {C[i][j]=C[i-1][j]; p[i][j]=p[i-1][j];}}
void afiseaza()
{int i=n,j=G,k; cout<<"Profit total= "<<C[n][G]<<endl;
 while (p[i][j]!=0)
 {k=p[i][j];
  cout<<"Obiectul "<<k<<" cu greutatea "<<a[k].g;
  cout<<" si profitul "<<a[k].c<<endl;
  j-=a[p[i][j]].g;
  while (p[i][j]==k) i--;}
 void main()
 {citeste(); p_dinamica(); afiseaza();}
```

Complexitatea algoritmului programării dinamice

Algoritmul programării dinamice are complexitatea **O(nxm)** deoarece se rezolvă cele **n** subprobleme, iar pentru fiecare subproblemă se calculează cele **m** elemente ale valorii asociate soluției.

-  **Temă**
- Se dau **n** numere naturale și un număr natural **S**. Să se găsească, dacă există, soluția de a obține numărul **S** ca sumă de numere naturale dintre cele **n** numere date.
 - Să se afișeze cel mai mic număr cu exact **n** divizori.
 - Se citesc de la tastatură două siruri de caractere **S₁** și **S₂**. Să se insereze spații în sirul **S₁** astfel încât dacă se suprapun cele două siruri, numărul de caractere care coincid să fie maxim.
 - Se dau două siruri de caractere **A** și **B** de lungime **n**, respectiv **m**. Într-un sir de caractere se pot executa numai următoarele trei operații prin care se poate transforma un sir: **S(i)** – sterge caracterul din poziția **i**, **I(i,c)** – inserează caracterul **c** în poziția **i** și **M(i,c)** – înlocuiește caracterul din poziția **i** cu caracterul **c**. Să se transforme sirul **A** în sirul **B** folosind un număr minim de operații. Problema se va rezolva în două variante: **a)** transformările se vor aplica de la stânga la dreapta; **b)** transformările se vor aplica de la dreapta la stânga.

5. Se dau două şiruri de numere $X=\{x_1, x_2, \dots, x_m\}$ și $Y=\{y_1, y_2, \dots, y_m\}$. Să se determine cel mai lung subşir comun al celor două şiruri. Şirul $Z=\{z_1, z_2, \dots, z_k\}$ este **subşir comun** al şirurilor X și Y dacă el este subşir al şirului X și subşir al şirului Y . De exemplu, dacă $X=\{1,2,3,2,2,1,4\}$ și $Y=\{2,1,3,1,2,4\}$, şirul $Z=\{2,3,1,4\}$ este un subşir comun al celor două şiruri. **Indicație.** Valoarea asociată soluției este lungimea unui subşir comun, iar funcția recursivă definește valoarea lungimii maxime a unui subşir comun $L(i,j)$ unde i este indicele elementului cu care începe subşirul X , iar j este indicele elementului cu care începe subşirul Y .

$$L(i,j) = \begin{cases} 0 & \text{dacă } i=0 \text{ sau } j=0 \\ 1+L(i-1,j-1) & \text{dacă } i \neq 0, j \neq 0 \text{ și } x_i = y_j \\ \max(L(i,j-1), L(i-1,j)) & \text{dacă } i \neq 0, j \neq 0 \text{ și } x_i \neq y_j \end{cases}$$

6. Se dă un sir de n matrice A_1, A_2, \dots, A_n și trebuie să se calculeze produsul lor: $A_1 \times A_2 \times \dots \times A_n$. Fiecare matrice A_i , cu $2 \leq i \leq n$, are p_{i-1} linii și p_i coloane (două matrice A și B se pot înmulți numai dacă numărul de coloane din matricea A este egal cu numărul de linii din matricea B). Să se pună în acest produs parantezele potrivite astfel încât numărul de înmulțiri să fie minim (deoarece înmulțirea matricelor este asociativă, inserarea parantezelor nu modifică rezultatul).

Indicație. Fiind date două matrice $A_{p,q}$ și $B_{q,r}$, matricea produs $C_{p,r}$ conține elementele:

$$c_{ij} = \sum_{k=1}^q a_{ik} \times b_{kj}$$

Timpul necesar pentru calculul matricei produs este determinat de numărul de înmulțiri scalare: $a_{ik} \times b_{kj}$. De exemplu, pentru produsul $A_1 \times A_2 \times A_3$, în care matricele au dimensiunile 10×20 , 20×5 și, respectiv, 5×40 , numărul de înmulțiri scalare pentru paranterizarea $((A_1 \times A_2) \times A_3)$ este $10 \times 20 \times 5 + 10 \times 5 \times 40 = 1200$, iar pentru paranterizarea $(A_1 \times (A_2 \times A_3))$ este $10 \times 20 \times 40 + 20 \times 5 \times 40 = 12000$ (de 10 ori mai mare decât în exemplul anterior). Valoarea asociată soluției este numărul de înmulțiri scalare, iar funcția recursivă $m(i,j)$ definește numărul minim de înmulțiri scalare pentru calculul matricei produs $A_1 \times \dots \times A_j$.

$$m(i,j) = \begin{cases} 0 & \text{dacă } i=j \\ \min\{ m(i,k) + m(k+1,j) + p_{i-1} \times p_k \times p_j \mid i \leq k < j \} & \text{dacă } i < j \end{cases}$$

Numărul minim de înmulțiri scalare pentru calculul produsului $A_1 \times \dots \times A_n$ este $m(1,n)$. Pentru a memora pozițiile parantezelor optime se va folosi matricea s , în care elementul s_{ij} are valoarea k ce reprezintă poziția parantezei optime în produsul $A_1 \times \dots \times A_j$. Altfel spus, s_{ij} este egal cu k pentru care $m(i,j) = m(i,k) + m(k+1,j) + p_{i-1} \times p_k \times p_j$.

1.7. Compararea metodelor de construire a algoritmilor

Metodele **backtracking**, **greedy** și **programarea dinamică** se recomandă în cazul problemelor care au următoarele caracteristici:

- soluția poate fi dată de un vector sau o matrice cu elemente de tip întreg;
- elementele soluției aparțin unui subdomeniu limitat.

Algoritmul backtracking are cea mai mare complexitate (complexitatea exponentială), iar algoritmul greedy, cea mai mică. Alegerea uneia dintre aceste metode se face în funcție de cerințele problemei:

- Dacă se cere determinarea tuturor soluțiilor, se va alege metoda **backtracking**.

→ Dacă se cere determinarea **soluției optime**, se va alege metoda **greedy**, dacă se poate demonstra că ea furnizează un rezultat corect; altfel, se va alege metoda **programării dinamice**.

Metodele greedy și programarea dinamică se folosesc în problemele în care trebuie găsită soluția optimă. În ambele metode alegerea elementului x_k al soluției este irevocabilă și poate duce la obținerea soluției optime în cazul problemelor care au **proprietatea de optim local**, adică soluția optimă a problemei cu dimensiunea n a datelor de intrare **conține soluțiile optime ale subproblemelor** similare cu problema inițială, dar de dimensiune mai mică. Deosebirea dintre cele două metode constă în modul în care este construită soluția:

1. În cazul metodei **programării dinamice**, se pornește de la structura problemei inițiale care corespunde soluției optime globale și se descompune această problemă în subprobleme în care se determină soluțiile optime ale subproblemelor (optimul local). Ea se poate aplica în problemele în care **optimul global implică optimul local**. Din această cauză, etapa cea mai dificilă la această metodă este cea de determinare a structurii soluției optime. Dacă se cunoaște structura soluției optime, metoda **garantează obținerea soluției optime** a problemei prin determinarea optimului local.
2. În cazul metodei **greedy**, se pornește de la soluțiile optime locale și pe baza lor se construiește soluția optimă globală. Problema este descompusă în subprobleme și soluția este construită treptat: se pornește de la mulțimea vidă și se alege pe rând pentru fiecare element al soluției candidatul optim pentru acea subproblemă. Alegerea soluțiilor optime locale **nu garantează** însă că soluția globală obținută este și soluția optimă, deoarece **optimul local nu implică întotdeauna optimul global**. Din această cauză, pentru a fi siguri că soluția aleasă este soluția optimă, pentru fiecare problemă trebuie să se demonstreze că modul de alegere a candidatului optim (optimul local) implică optimul global. Metoda greedy este mai simplă decât metoda programării dinamice deoarece la fiecare pas se tratează numai subproblemă curentă, iar ordinul de complexitate este mai mic decât în cazul programării dinamice.

Metodele greedy și backtracking construiesc treptat soluția problemei. Deosebirea dintre cele două metode constă în modul în care se construiește soluția. În cazul metodei **greedy**, când se alege elementul x_k al soluției, dacă elementul a_i care este candidatul optim al soluției nu este element al soluției, el este îndepărtat din mulțimea elementelor candidate la soluție (A). În acest mod el nu va mai fi verificat atunci când se alege un alt element al soluției, micșorând foarte mult ordinul de complexitate al algoritmului.

Pentru a obține soluția dorită folosind un algoritm cât mai eficient se pot **combina metodele** învățate.

Exemplu. Problema **plății unei sume cu un număr minim de bancnote cu valori date**. Problema a fost rezolvată cu ajutorul metodei **greedy**. Folosind această metodă este posibil să nu obțineți nicio soluție, sau soluția obținută să nu fie soluția optimă. Problema poate fi rezolvată combinând metoda **greedy** (prin ordonarea vectorului cu valorile bancnotelor după criteriul candidatului optim) cu metoda **backtracking**. Soluția optimă va fi printre primele soluții generate. Apelul subprogramului **bt()** se va opri imediat ce se depășește numărul minim de bancnote. Prin combinarea celor două metode se obține un algoritm mai eficient decât algoritmul **backtracking**, care va permite găsirea soluției optime.

Metodele divide et impera și programarea dinamică se bazează pe descompunerea problemei în subprobleme similare cu problema inițială. Deosebirile dintre cele două metode sunt:

Divide et impera

1. Subproblemele

Subproblemele sunt independente și nu se cunosc de la început subproblemele care apar în urma descompunerii. De multe ori descompunerea în subprobleme depinde de distribuția datelor de intrare și este posibil ca în aceeași problemă, pentru date de intrare diferite, să se obțină descompuneri diferite (de exemplu, algoritmul căutării binare sau algoritmul sortării rapide).

2. Asigurarea eficienței metodei

Este eficientă numai dacă este respectată regula de descompunere a problemei în subprobleme independente. Dacă nu este asigurată independența problemelor, metoda devine neficientă deoarece, în apelurile recursive, aceeași subproblemă este rezolvată de mai multe ori.

3. Modul de descompunerea în subprobleme

Descompunerea se face într-o manieră „de sus în jos” („top-down”): problema inițială este descompusă în subprobleme din ce în ce mai mici până se obțin subprobleme cu rezolvare imediată, se rezolvă aceste subprobleme, după care se combină soluțiile subproblemelor de dimensiuni din ce în ce mai mari.

4. Algoritm

Algoritm este **recursiv**.

Din cauza acestor deosebiri aceste metode se folosesc pentru clase diferite de probleme. De exemplu, **problema determinării termenului n al sirului lui Fibonacci**. Rezolvarea acestei probleme prin metoda divide et impera este **neeficientă** deoarece pentru a calcula termenul f_n trebuie să se calculeze termenii f_{n-1} și f_{n-2} . Dar, în calculul termenului f_{n-1} , reappeare calculul termenului f_{n-2} , iar în calculul ambilor termeni apare calculul termenului f_{n-3} și.m.d. În acest caz, se recomandă metoda programării dinamice, calculând pe rând termenii f_3, f_4, \dots, f_n , pornind de la termenii $f_1=f_2=1$. Formula de recurență este: $f_i = f_{i-1}+f_{i-2}$, pentru $3 \leq i \leq n$. Deoarece termenul f_i nu depinde decât de doi termeni f_{i-1} și f_{i-2} , nu este necesar să se memoreze întregul sir de termeni, ci pentru fiecare subproblemă i , numai termenii obținuți în subproblemele $i-1$ și $i-2$.

Programarea dinamică

Subproblemele **nu sunt independente** și în descompunerea inițială există subprobleme comune unor subprobleme de dimensiuni mai mari. Pentru a se putea rezolva problema trebuie să se cunoască, de la început, **subproblemele** în care se descompune problema inițială. Aceasta presupune cunoașterea structurii problemei inițiale și a relației de recurență care determină soluția.

Este eficientă numai dacă o subproblemă este rezolvată o singură dată, soluția sa este memorată și folosită pentru a obține soluțiile celorlalte subprobleme.

Descompunerea se face într-o manieră „de jos în sus” („bottom-up”): se rezolvă problemele de dimensiunile cele mai mici care apar în problema inițială, se memorează soluțiile lor și se rezolvă subproblemele de dimensiuni din ce în ce mai mari prin combinarea soluțiilor problemelor mai mici și memorarea noilor soluții.

Algoritmul este **iterativ**.

Evaluare

Adevărat sau Fals:

- În algoritmul metodei backtracking inițializarea elementului k al soluției se face când se revine de pe nivelul $k+1$ la nivelul k .

2. Algoritmul metodei backtracking se încheie dacă s-au testat toate valorile posibile pentru primul element al soluției.
3. În algoritmul backtracking, după găsirea unei soluții, se revine la nivelul anterior al soluției.
4. În algoritmul backtracking, trecerea de la nivelul k la nivelul $k-1$ al soluției se face după ce s-au găsit toate valorile posibile pentru nivelul k .
5. În algoritmul backtracking inițializarea nivelului k al soluției se face cu valoarea de pe nivelul anterior al soluției.
6. În algoritmul backtracking, dacă s-a trecut la nivelul k al soluției, înseamnă că s-au găsit primele $k-1$ elemente ale soluției.
7. În algoritmul backtracking, trecerea la nivelul $k+1$ al soluției se face după ce au fost testate toate valorile posibile pentru nivelul k al soluției.
8. În implementarea recursivă a metodei backtracking revenirea din autoapel este echivalentă cu revenirea la nivelul anterior al soluției din varianta iterativă.
9. În implementarea recursivă a metodei backtracking autoapelul este echivalent cu trecerea la nivelul următor al soluției din varianta iterativă.
10. Divide et impera este metoda prin care problema inițială se descompune în subprobleme care nu sunt independente.
11. Prin metoda divide et impera se găsește soluția optimă a problemei.
12. Metoda programării dinamice pornește de la soluțiile optime locale și, pe baza lor, construiește soluția optimă globală.
13. În metoda programării dinamice descompunerea în subprobleme se face „de sus în jos”.
14. Metoda greedy se bazează pe teorema că optimul local implică întotdeauna optimul global.

Alegeți:

1. Algoritmul care implementează metoda backtracking este:
 - polinomial
 - logaritmic
 - exponențial
 - liniar logaritmic
2. Algoritmul care implementează metoda sortării rapide este:
 - liniar
 - logaritmic
 - pătratic
 - liniar logaritmic
3. Algoritmul care implementează metoda căutării binare este:
 - liniar
 - logaritmic
 - pătratic
 - liniar logaritmic
4. Se utilizează metoda backtracking pentru generarea anagramelor cuvântului *masca*. Să se precizeze cuvântul precedent și cuvântul următor secvenței *camas*, *camsa*, *caams*:
 - csaam* și *casma*
 - csama* și *casma*
 - csaam* și *casma*
 - csaam* și *caasm*
5. Se utilizează metoda backtracking pentru generarea tuturor numerelor cu 4 cifre distincte care se pot forma cu cifrele 0, 2, 4 și 8. Să se precizeze numărul precedent și numărul următor secvenței 2840 4028 4082:
 - 2480 și 4280
 - 2804 și 4280
 - 2804 și 4208
 - 2480 și 4280
6. Se utilizează metoda backtracking pentru generarea sirului de opt paranteze rotunde astfel încât ele să formeze o succesiune corectă. Să se precizeze sirul de paranteze precedent și sirul de paranteze următor secvenței $((())(())\ (())()\ (())()$:
 - $((())(())$ și $(())()$
 - $(())()$ și $(())(())$
 - $(())()$ și $(())()$
 - $(())()$ și $(())()$
7. Se utilizează metoda backtracking pentru generarea submulțimilor multimii {1,2,3}. Să se precizeze submulțimea precedentă și submulțimea următoare secvenței de submulțimi {1} {1,3} {1,2}:
 - { } și {2,3}
 - { } și {2}
 - {2,3} și {1,2,3}
 - { } și {3}

8. Dacă se utilizează metoda backtracking pentru a genera toate permutările de 5 obiecte și primele 4 permutări generate sunt: 5 4 3 2 1, 5 4 3 1 2, 5 4 2 3 1, 5 4 2 1 3, atunci a cincea permutare este:
 a. 5 4 3 2 1 b. 5 3 4 2 1 c. 5 4 1 2 3 d. 5 4 1 3 2
 (Bacalaureat – Sesiunea specială 2003)
9. Se cere determinarea tuturor modalităților de planificare în zile diferite a 4 probe (rezistență, aruncarea sulișei, sărituri, tir) în oricare dintre cele 7 zile din săptămână. Problema este echivalentă cu:
 a. generarea combinărilor de 4 obiecte luate câte 7
 b. generarea aranjamentelor de 4 obiecte luate câte 7
 c. generarea combinărilor de 7 obiecte luate câte 4
 d. generarea aranjamentelor de 7 obiecte luate câte 4
 (Bacalaureat – Sesiunea specială 2003)
10. Generarea tuturor sirurilor de 3 elemente, fiecare element putând fi oricare număr din mulțimea {1,2,3,4,5,6}, se realizează cu ajutorul unui algoritm echivalent cu algoritmul de generare a:
 a. aranjamentelor b. permutărilor c. combinărilor d. produsului cartezian
 (Bacalaureat – Sesiunea specială 2003)
11. Se cere determinarea tuturor modalităților distincte de așezare în linie a tuturor celor n sportivi aflați la o festivitate de premiere. Problema este echivalentă cu generarea:
 a. partităilor unei mulțimi de n obiecte b. aranjamentelor de n obiecte luate câte 1
 c. permutărilor de n obiecte d. submulțimilor unei mulțimi cu n obiecte
 (Bacalaureat – Sesiunea iunie-iulie 2003)
12. Condiția ca două dame Q și D (de tip structură) să se afle pe aceeași diagonală a tablei de șah este:
 a. (Q.linie==D.linie) || (Q.coloana==D.coloana)
 b. abs(Q.linie-D.linie)==abs(Q.coloana-D.coloana)
 c. Q.linie-D.linie == Q.coloana-D.coloana
 d. abs(Q.linie-D.coloană)==abs(D.linie-D.coloana)
 (Bacalaureat – Sesiunea iunie-iulie 2003)
13. Se cere determinarea tuturor numerelor formate cu cifre aflate în ordine strict crescătoare, cifre alese dintr-o mulțime cu k cifre distincte date. De exemplu, pentru cifrele alese din mulțimea {1,5,2} se obțin numerele 1, 2, 12, 5, 15, 25, 125 (nu neapărat în această ordine). Problema este echivalentă cu generarea:
 a. partităilor unei mulțimi b. aranjamentelor de 10 obiecte luate câte k
 c. permutărilor de k obiecte d. submulțimilor nevide ale unei mulțimi
 (Bacalaureat – Sesiunea august 2003)
14. Cineva dorește să obțină și să prelucreze toate numerele formate din trei cifre din sirul 2, 9, 4 și le generează exact în ordinea 222, 229, 224, 292, 299, 294, 242, 249, 244, 922, 929, 924, 992, 999, 994, 942, 949, 944, 422, 429, 424, 492, 499, 494, 442, 449, 444. Dacă dorește să obțină prin același procedeu numerele formate din 4 cifre, atunci după numărul 4944 va urma:
 a. 4949 b. 9222 c. 4422 d. 4924
 (Bacalaureat – Simulare 2004)
15. Un elev folosește metoda backtracking pentru a genera submulțimile mulțimii {1,2,5,6,9}. Câte soluții (submulțimi) care obligatoriu conțin elementul 2 și nu conțin elementul 6 a generat?
 a. 7 b. 16 c. 6 d. 8
 (Bacalaureat – Sesiunea specială 2004)

16. Pentru a determina toate modalitățile de a scrie pe 9 ca sumă de numere naturale nenule distințe (abstracție făcând de ordinea termenilor), un elev folosește metoda backtracking generând, în această ordine, toate soluțiile: 1+2+6, 1+3+5, 1+8, 2+3+4, 2+7, 3+6 și 4+5. Aplicând exact aceeași metodă, el determină soluțiile pentru scrierea lui 12. Câte soluții de forma 3+... există?

- a. 7 b. 1 c. 2 d. 4

(Bacalaureat – Sesiunea iunie-iulie 2004)

17. Pentru problema anterioară, care este a opta soluție determinată?

- a. 2+3+7 b. 3+4+5 c. 1+5+6 d. 1+11

(Bacalaureat – Sesiunea iunie-iulie 2004)

18. Pentru problema anterioară stabiliți care este soluția cu proprietatea că imediat după ea este generată soluția 3+4+5?

- a. 3+4+6 b. 2+3+7 c. 2+10 d. 4+8

(Bacalaureat – Sesiunea iunie-iulie 2004)

19. Se generează toate permutările distințe de 4 obiecte numerotate de la 1 la 4, având proprietatea că 1 nu este vecin cu 3 (1 nu se află imediat după 3, și nici 3 imediat după 1) și 2 nu este vecin cu 4. Câte astfel de permutări se generează?

- a. 12 b. 24 c. 6 d. 8

(Bacalaureat – Sesiunea august 2004)

20. Se cere să se determine toate modalitățile distințe de a programa n spectacole în z zile (de exemplu, două spectacole „Micul Print” și „Păcală” se pot programa în trei zile, marți, miercuri și joi, astfel: „Micul Print” – marți și „Păcală” – miercuri sau; „Micul Print” – miercuri și „Păcală” – marți; „Micul Print” – miercuri și „Păcală” – joi; „Micul Print” – marți și „Păcală” – joi; „Micul Print” – joi și „Păcală” – miercuri etc.). Rezolvarea se bazează pe algoritmul de generare a:

- a. combinărilor de n obiecte luate câte z b. aranjamentelor de n obiecte luate câte z
c. combinărilor de z obiecte luate câte n d. aranjamentelor de z obiecte luate câte n

(Bacalaureat – Simulare 2005)

21. Se consideră algoritmul care determină toate permutările distințe de n obiecte (numerotate de la 1 la n) în care nu există poziții fixe. O permutare (p_1, p_2, \dots, p_n) are puncte fixe dacă există cel puțin o componentă $p_i=i$. De exemplu, pentru $n=5$, permutarea $(2,3,5,4,1)$ are puncte fixe deoarece $p_4=4$. Pentru $n=4$, stabiliți câte permutări fără puncte fixe există.

- a. 8 b. 9 c. 10 d. 12

(Bacalaureat – Sesiunea specială 2005)

22. Se consideră algoritmul care determină toate permutările distințe de n obiecte (numerotate de la 1 la n) în care pe orice poziție de rang par se află o valoare pară. De exemplu, pentru $n=5$, primele trei permutări generate în ordine lexicografică sunt $(1,2,3,4,5)$, $(1,2,5,4,3)$, $(1,4,3,2,5)$. Pentru $n=4$, numărul total de astfel de permutări este:

- a. 2 b. 4 c. 8 d. 12

(Bacalaureat – Sesiunea iunie-iulie 2005)

23. Se consideră algoritmul care generează, în ordine strict crescătoare, toate numerele formate din n cifre distințe (cifrele de la 1 la n), numere în care pe fiecare poziție pară se afă o cifră impară. Poziția se numără de la stânga către dreapta, pornind de la poziția 1. De exemplu, pentru $n=4$, primele trei numere generate sunt: 2143, 2341, 4123. Pentru $n=5$, primul număr generat este:

- a. 21345 b. 21435 c. 12345 d. 13254

(Bacalaureat – Sesiunea august 2005)

24. Se generează toate submulțimile formate din două elemente ale mulțimii {5,6,7,8} în ordinea: 5 6, 5 7, 5 8, 6 7, 6 8, și 7 8. Dacă se utilizează exact aceeași metodă pentru a genera submulțimile de trei elemente ale mulțimii {2,3,4,5,6}, atunci penultima mulțime generată este:

a. 3 4 5

b. 3 5 6

c. 4 5 6

d. 2 5 6

(Bacalaureat – Simulare 2006)

Miniproiecte:

Pentru realizarea **miniproiectelor** se va lucra **în echipă**. Fiecare miniproiect va conține:

- două metode pentru construirea algoritmului (care sunt precizate în cazul primelor 8 miniproiecte);
- justificarea folosirii metodelor care au fost alese pentru rezolvarea problemei;
- explicarea metodelor folosite pentru construirea algoritmului;
- compararea celor două metode din punct de vedere al corectitudinii soluției;
- compararea celor doi algoritmi din punct de vedere al complexității.

- Rezolvați problema acoperirii tablei de șah de dimensiunea $n \times n$ cu un cal care pornește din pătratul de coordonate (x,y) folosind metoda backtracking și metoda greedy.
- Rezolvați problema costruirii țevii de lungime L din bucăți de țeavă de lungimi a_i folosind metoda backtracking și metoda programării dinamice.
- Rezolvați problema găsirii unui traseu pentru a ieși din labirintul de dimensiunea $n \times m$, știind că se pornește din pătratul de coordonate (x,y) , folosind metoda greedy și metoda backtracking.
- Rezolvați problema plății unei sume cu bancnote cu valori date, pentru fiecare valoare de bancnotă având la dispoziție un anumit număr de bancnote, folosind metoda programării dinamice și metoda backtracking.
- Rezolvați problema turistului care trebuie să găsească un traseu de munte în care să urce cât mai puțin, folosind metoda programării dinamice și metoda greedy.
- Rezolvați problema determinării sumei maxime într-un triunghi folosind metoda greedy și metoda programării dinamice.
- Rezolvați problema determinării unui subșir crescător de lungime maximă folosind metoda greedy și metoda programării dinamice.
- Se dau n paralelipipede având fiecare dimensiunile a_i , b_i și c_i . Să se construiască un turn de înălțime maximă prin suprapunerea acestor paralelipipede. Un paralelipiped poate fi aşezat pe orice față. Un paralelipiped poate fi aranjat peste un alt paralelipiped numai dacă fața lui nu depășește fața paralelipipedului pe care este pus. Rezolvați problema folosind metoda backtracking și metoda programării dinamice.
- Pe o tablă de șah de dimensiunea $n \times n$ un cal pornește din pătratul de coordonate (x_1, y_1) și trebuie să ajungă în pătratul de coordonate (x_2, y_2) . Să se găsească traseul de lungime minimă.
- Se dau n piese de domino, fiecare piesă fiind descrisă prin perechi de două numere (x_i, y_i) . Să se găsească o secvență de lungime maximă de piese în care oricare două piese alăturate au înscrise la capetele cu care se învecinează același număr.

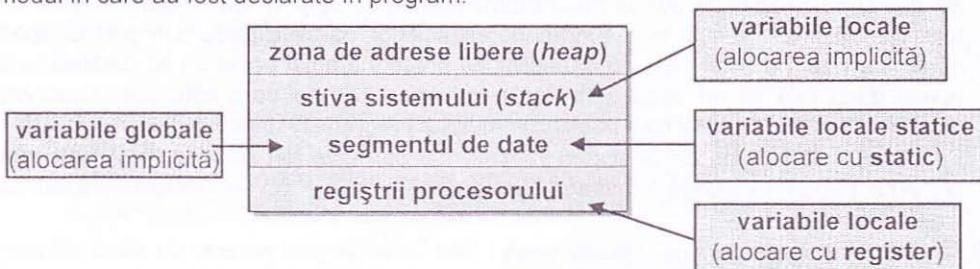
2. Implementarea structurilor de date

2.1. Tipuri de date specifice pentru adresarea memoriei

Memoria internă a calculatorului este organizată sub forma unor locații de dimensiunea unui octet, numerotate consecutiv, pornind de la 0. Aceste numere, exprimate în hexazecimal, se numesc **adrese de memorie**. Locațiile de memorie pot fi manipulate individual sau în grupuri contigue. **Spațiul de memorie este gestionat de sistemul de operare.** Fiecare program aflat în execuție, sistemul de operare îl rezervă (alocă) propriul spațiu de memorie. Dacă programul apelează subprograme, acestea pot fi privite ca blocuri care conțin instrucțiuni și date (variabile de memorie și structuri de date), cărora sistemul de operare trebuie să le aloce spațiu de memorare în interiorul zonei de memorie rezervată programului principal (care corespunde fișierului sursă). Spațiul de memorare este împărțit în patru segmente, iar procesorul are acces la toate cele patru segmente, ale căror adrese le gestionează prin intermediul unor registri:

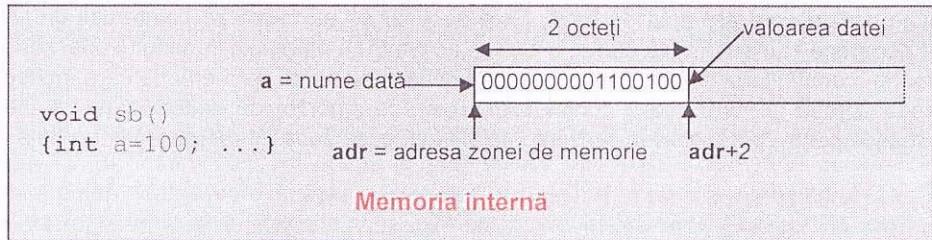
- **segmentul de cod** – în care se încarcă instrucțiunile programului care se execută; adresa de început se păstrează în registrul CS;
- **segmentul de date** – în care se încarcă anumite date ale programului care se execută (de exemplu, variabilele globale); adresa de început se păstrează în registrul DS;
- **segmentul de stivă a sistemului** – în care se încarcă adresele de revenire din subprograme și variabilele locale cu care lucrează subprogramele; pentru gestionarea sa se folosesc doi registri: SS (în care se păstrează adresa de la baza stivei) și SP (în care se păstrează adresa de la vârful stivei);
- **extrasegmentul de date** – în care se încarcă alte date ale programului; adresa de început se păstrează în registrul ES.

Sistemul de operare alocă spațiu datelor în mai multe **zone de memorie**, în funcție de modul în care au fost declarate în program.



O dată manipulată de program prin intermediul unei variabile de memorie este caracterizată de mai multe **attribute** – pe care le primește în momentul în care este declarată în program:

- **Numele**: Este identificatorul folosit pentru referirea datei în cadrul programului (în exemplu – a).
- **Adresa**: Este adresa de memorie internă la care se alocă spațiu datei respective, pentru a stoca valoarea datei (în exemplu, datei i se alocă adresa adr în stiva sistemului).

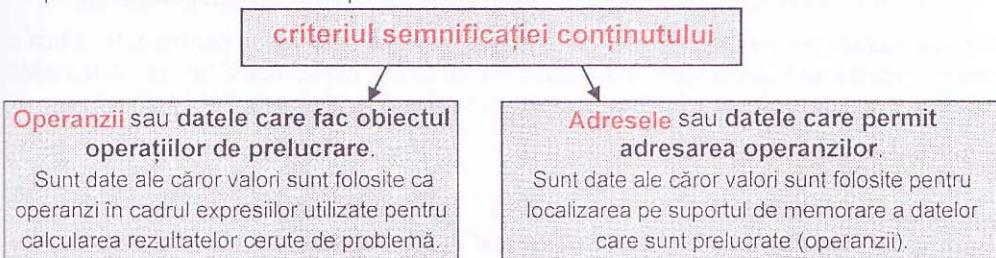


- **Valoarea**. Este conținutul, la un moment dat, al zonei de memorie rezervate datei (în exemplu – $100_{(10)} = 1100100_{(2)}$).
- **Tipul**. Determină: **domeniul de definiție intern** al datei (mulțimea în care poate lua valori date), **operatorii** care pot fi aplicati pe acea dată și **modul** în care data este **reprezentată în memoria internă** – metoda de codificare în binar a valorii datei (în exemplu, tipul este **unsigned int**: domeniul de definiție intern al datei este intervalul [0, 65535]), operatorii care pot fi aplicati pe această dată sunt operatorii permisi de tipul numeric – aritmetici, relaționali, logici, logici pe biți etc. – și reprezentarea datei în memoria internă se face prin conversia numărului din decimal în binar).
- **Lungimea**. Este dimensiunea zonei de memorie alocate datei și se măsoară în octeți. Ea depinde de modul de reprezentare internă a tipului de dată (în exemplu, datei i se alocă un grup contigu de **2 octeți**). Pentru a afla dimensiunea zonei de memorie alocate unei variabile de memorie se poate folosi operatorul **sizeof**.
- **Durata de viață**. Este perioada de timp în care variabilei i se alocă spațiu de memorare (în exemplu, data a este o **variabilă cu durată locală** căreia i se alocă spațiu de memorare numai pe durata de execuție a blocului în care a fost declarată, adică pe durata de execuție a subprogramului **sb**).

Pentru a putea manipula o dată, programul trebuie să identifice zona de memorie în care este stocată. Identificarea acestei zone se poate face atât prin numele datei (a), cât și prin adresa la care este memorată (adr). Pentru a putea manipula datele cu ajutorul adreselor de memorie, în limbajul C++ sunt implementate următoarele tipuri de date:

- tipul **pointer** sau **adresă**,
- tipul **referință**.

Se poate folosi un nou criteriu pentru **clasificarea datelor**:



2.2. Tipul de dată pointer

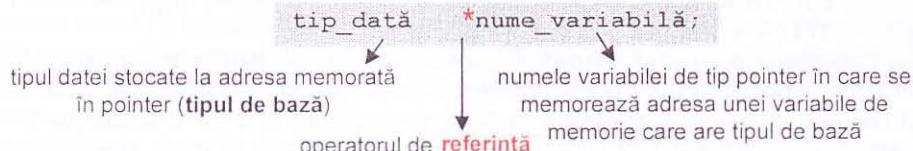
Pointerul este o variabilă de memorie în care se memorează o **adresă de memorie**.

Observație. Un pointer se asociază întotdeauna unui **tip de dată** (de exemplu: **int**, **char**, **float** etc.) numit **tip de bază**. Se spune că un pointer indică o zonă de memorie care

conține o dată care are tipul de bază. Este necesar să se asocieze pointerului un tip de dată deoarece el memorează numai o adresă și, pentru a manipula conținutul unei zone de memorie, compilatorul trebuie să cunoască dimensiunea zonei de memorie care începe de la acea adresă și semnificația conținutului, adică ce algoritm de decodificare trebuie să folosească pentru a transforma secvența de biți (din acea zonă de memorie) într-o dată.

2.2.1. Declararea variabilelor de tip pointer

Declararea unei variabile de tip pointer se poate face prin instrucțiunea declarativă:



Exemplu:

```
char *p;
int *q;
float *r;
```

S-au definit trei variabile de tip adresă (pointeri): **p** către tipul **char**, **q** către tipul **int** și **r** către tipul **float**. Numele acestor variabile de memorie sunt **p**, **q** și **r**, iar tipul de bază al fiecărei variabile de tip pointer este **char**, **int**, respectiv **float**. Așadar, **p** este o variabilă de tip adresă a unei locații ce conține un caracter care ocupă 1 octet, **q** este o variabilă de tip adresă a unei locații ce conține o dată numerică întreagă care ocupă 2 octeți, iar **r** este o variabilă de tip adresă a unei locații ce conține o dată numerică reală care ocupă 4 octeți.

Se poate declara un **pointer către tipul de dată înregistrare** (adresa unei înregistrări).

Exemplu:

```
struct punct {int x,y;};
punct *p;
```

S-a definit variabila **p** de tip adresă către tipul înregistrare **punct**, adică o adresă a unei locații care conține două date numerice întregi de tip **int** ce ocupă 4 octeți.

Pointerul zero are valoarea 0 și indică zona de memorie de la adresa 0.

Pentru limbajul C++ adresa 0 înseamnă o adresă care nu este validă pentru date, adică o adresă **inexistentă**. Altfel spus, pointerul zero nu indică nimic. Acest tip de pointer este folosit ca **terminator** în structurile de date care sunt prelucrate cu ajutorul pointerilor.

2.2.2. Constante de tip adresă

O **constantă de tip adresă** este un pointer al cărui conținut nu poate fi modificat.

Exemplu. **Constanta 0** – care semnifică „adresă nulă” (adresă nonexistentă sau adresă nealocată) – este utilă, în unele cazuri, pentru inițializarea valorii unui pointer. În locul constantei numerice literale 0, se poate folosi **constantă simbolică predefinită NULL**.

Observație. La declararea unui **tablou de memorie**, acestuia i se alocă o zonă de memorie de dimensiune fixă, începând de la o anumită adresă de memorie. **Adresa de la care se alocă zona de memorie este o constantă** (nu poate fi modificată) și ea este asociată cu

numele tabloului de memorie. **Numele tabloului de memorie** este folosit de compilatorul C++ ca o **constantă simbolică de tip adresă** (pointer către tipul de bază dat de tipul elementelor tabloului).

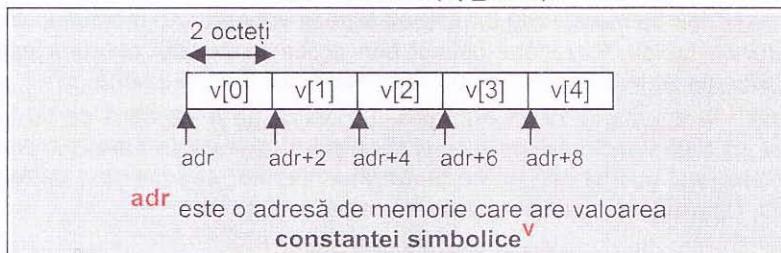
Tabloul de memorie este o structură de date omogenă, adică o colecție de elemente de același tip, și fiecare element ocupă același număr de octeți. Acest mod de memorare permite determinarea adresei fiecărui element pornind de la adresa simbolică a tabloului, care va reprezenta și adresa primului element. Identificarea unui element de tablou de memorie se face folosind pentru fiecare element un grup de indici (numărul de indici fiind egal cu dimensiunea tabloului), adresa elementului calculându-se față de un element de referință care este adresa tabloului. Deoarece adresa tabloului este și adresa primului element, în limbajul C++ **numerotarea indicilor se face pornind de la 0**, indicele reprezentând deplasarea elementului față de adresa tabloului.

Exemplul 1:

```
int v[5];
```

Zona de memorie alocată unui **vector** are dimensiunea $n \times \text{sizeof}(\text{tip_bază})$. De exemplu, vectorului **v** i se va aloca o zonă de memorie de $5 \times \text{sizeof}(int) = 5 \times 2 = 10$ octeți. Identificarea unui element de tablou de memorie se face folosind un singur indice (**i**). Dacă adresa tabloului **v** este **adr**, ea este și adresa elementului **v[0]**, iar adresa elementului **v[i]** este:

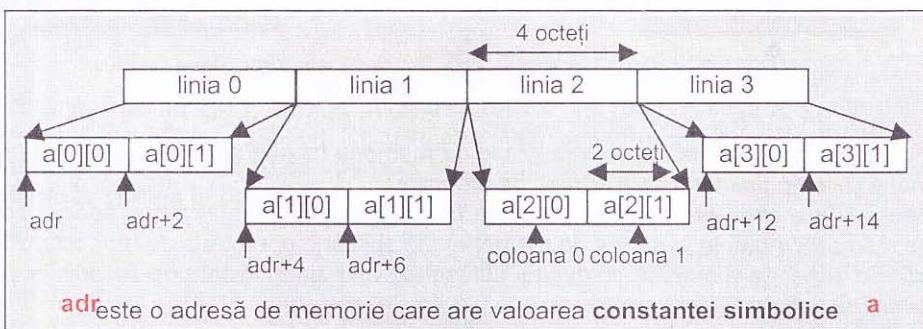
$\text{adr} + i \times \text{sizeof}(\text{tip_bază})$.



Pentru exemplul de mai sus, adresa elementului **v[2]** este:

$$\text{adr} + 2 \times \text{sizeof}(int) = \text{adr} + 2 \times 2 = \text{adr} + 4.$$

Exemplul 2:



Alocarea memoriei unei **matrice** se face în mod contiguu, memorarea elementelor făcându-se linie cu linie, astfel încât matricea apare ca un vector de linii. Dacă matricea are **m** linii și **n** coloane, ea va avea $m \times n$ elemente. De exemplu, matricea **a** are $4 \times 2 = 8$ elemente și este memorată ca un vector cu **m** elemente de tip vector cu **n** elemente, care au tipul de bază al matricei (în exemplu, un vector cu 4 elemente de tip vector cu 2 de elemente de tip **int**). Zona

de memorie alocată unei matrice are dimensiunea $m \times n \times \text{sizeof}(\text{tip_bază})$. În exemplu, matricei a se alocă o zonă de memorie de $4 \times 2 \times \text{sizeof}(\text{int}) = 4 \times 2 \times 2 = 16$ octeți. Identificarea unui element al matricei se face folosind doi indici (i și j). Adresa tabloului este și adresa primului element, $a[0][0]$. Dacă adresa tabloului este adr , adresa elementului $a[i][j]$ este:

$$\text{adr} + i \times n \times \text{sizeof}(\text{tip_bază}) + j \times \text{sizeof}(\text{tip_bază}) = \text{adr} + (i \times n + j) \times \text{sizeof}(\text{tip_bază}).$$

Pentru exemplul de mai sus, adresa elementului $a[3][1]$ este:

$$\text{adr} + 3 \times 2 \times \text{sizeof}(\text{int}) + 1 \times \text{sizeof}(\text{int}) = \text{adr} + 3 \times 2 \times 2 + 1 \times 2 = \text{adr} + 14.$$

2.2.3. Operatori pentru variabile de tip pointer

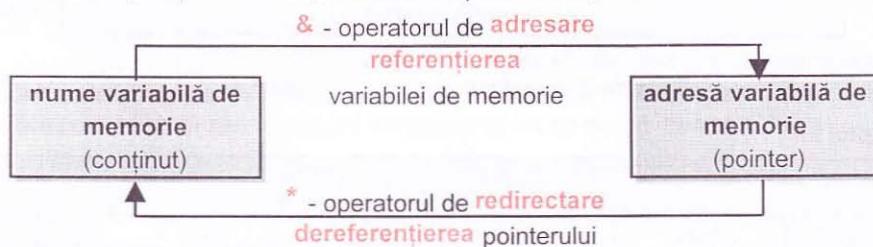
Pe variabilele de tip pointer se pot aplica următoarele tipuri de operatori:

- operatori specifici,
- operatorul de atribuire,
- operatori aritmetici,
- operatori relaționali.

2.2.3.1. Operatorii specifici

Operatorii specifici variabilelor de tip pointer sunt:

- **Operatorul &** – operatorul de **adresare**. Se aplică pe o variabilă de memorie sau un element de tablou de memorie și furnizează adresa variabilei de memorie, respectiv a elementului de tablou. Rezultatul obținut prin aplicarea acestui operator este de tip pointer. Operația se numește **referențierea** unei variabile de memorie.
- **Operatorul *** – operatorul de **redirectare**. Se aplică pe o variabilă de tip pointer și furnizează valoarea variabilei de memorie care se găsește la adresa memorată în pointer. Rezultatul obținut prin aplicarea acestui operator este de tipul datei asociate pointerului. Operația se numește **dereferențierea** unui pointer.



Observații:

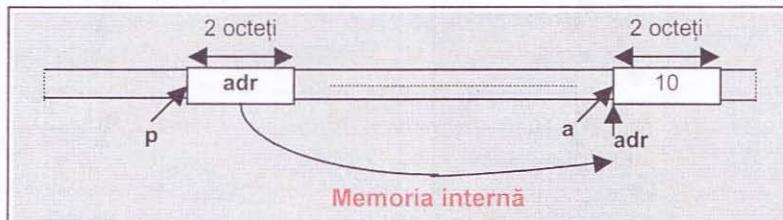
1. Pointerii reprezintă adrese ale unei zone de memorie (conținutul unei variabile de memorie p de tip pointer este o adresă de memorie).
2. Accesul la o zonă de memorie se poate face fie folosind identificatorul asociat zonei de memorie (numele variabilei de memorie), fie aplicând operatorul de redirectare * pe adresa zonei de memorie (conținutul adresei de memorie indicate de un pointer p se referă cu $*p$).
3. Dimensiunea și semnificația conținutului unei zone de memorie indicate de un pointer depind de tipul pointerului.

Exemplul 1

```

int a=10, *p=&a;
cout<<*p<<endl<<a<<endl;
cout<<p<<endl<<&a<<endl;
    
```

S-a definit o variabilă de memorie de tip **int** (**a**) și o variabilă de tip pointer către tipul **int** (***p**) căreia i s-a atribuit ca valoare adresa variabilei **a**. În memoria internă se vor aloca două zone de memorie: una de 2 octeți, pentru variabila de memorie identificată prin numele **a**, în care se păstrează o valoare numerică întreagă, și una de 2 octeți pentru variabila pointer identificată prin numele **p**, în care se păstrează o adresă de memorie (în exemplu, adresa variabilei de memorie **a**). Conținutul acestor zone de memorie este prezentat în figură.



Prin instrucția `cout<<*p;` se afișează conținutul variabilei de memorie **a**. Referirea la această variabilă de memorie se face nu prin numele variabilei, ci prin operatorul de redirecțare ***** aplicat pe variabila pointer **p** în care este memorată adresa variabilei **a**. Această instrucție are același efect ca și instrucția `cout<<a;`. Prin instrucția `cout<<p;` se afișează o constantă hexazecimală care reprezintă o adresă de memorie (adresa **adr** la care este memorată variabila **a**). Această instrucție are același efect ca și instrucția `cout<<&a;`.

Exemplul 2

```
int a=10,b=20,*p=&a;
b=*p;           /*variabilei b i se atribuie valoarea de la adresa
memorată în variabila p, adică valoarea variabilei a */
cout<<a<<" "<<b;           //afișează 10 10
*p=100;          /*variabilei a cărei adresă este memorată în
pointerul p (adică, variabilei a) i se atribuie valoarea 100 */
cout<<a<<" "<<b;           //afișează 100 10
```

Observație. Dacă tipul de dată indicat de pointer este **tipul înregistrare**, data conține mai multe câmpuri. Operatorul de redirectare prin care se furnizează conținutul unui câmp este operatorul **->** și se numește **operatorul de selecție indirectă a membrului unei structuri**.

`<nume_variabilă_pointer> -> <nume_câmp>`

Ei conține două referințe: una la conținutul adresei indicate de pointer și alta către un membru al înregistrării (câmpul). Rezultatul furnizat de expresie este valoarea membrului selectat (a câmpului). Este un **operator binar** care are **prioritate maximă** și **asociativitatea de la stânga la dreapta**.

Exemplu

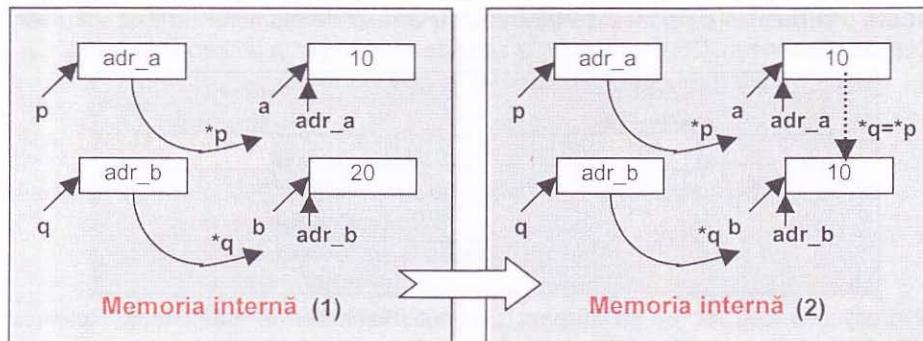
```
struct punct {int x,y;};
punct a={3,4},*p=&a;
cout<<a.x<<" "<<a.y<<endl;           //afișează 3 4
cout<<p->x<<" "<<p->y;           //afișează 3 4
```

Observații:

1. Operatorul ***** se poate folosi în ambeii membri ai unei operații de atribuire:

```
int a=10,b=20,*p=&a,*q=&b;           // (1)
*q=*p;                                // (2)
```

```
/* variabilei de la adresa memorată în variabila q, adică variabilei b, i se atribuie valoarea de la adresa memorată în variabilă p, adică valoarea variabilei a
cout<<a<<" " <<b; //afisează 10 10
```



2. Construcția ***p** (p fiind un pointer către un tip de dată) poate apărea în orice expresie în locul unei variabile de memorie care are același tip ca și cel asociat pointerului:

```
int a=10,*p=&a;
*p=*p+5; /* incrementează cu 5 valoarea variabilei de la adresa memorată în variabila p, adică valoarea variabilei a*/
cout<<a<<" "<<*p; //afisează 15 15
```

3. Operatorii ***** și **&** sunt operatori unari și au **prioritate mai mare** decât operatorii aritmetici, relaționali, logici și de atribuire.

```
int a=10,*p=&a;
*p+=1; /* incrementează cu 1 valoarea variabilei de la adresa memorată în p, adică valoarea variabilei a */
cout<<a; //afisează 11
```

4. **Asociativitatea** operatorilor unari este de la dreapta la stânga. Din această cauză, trebuie folosiți cu grijă operatorii unari ***** și **++** (respectiv **--**). Astfel, ***p++** incrementează cu 1 adresa memorată în variabila p, și nu valoarea de la adresa memorată în p:

```
int a=10,*p=&a;
**p; /* incrementează cu 1 valoarea variabilei de la adresa memorată în p, adică valoarea variabilei a */
cout<<a; //afisează 11
(*p)++; /* incrementează cu 1 valoarea variabilei de la adresa memorată în p, adică incrementează cu 1 valoarea variabilei a */
cout<<a; //afisează 12
```

5. Datorită tehnicii de supraîncărcare a operatorilor permisă de limbajul C++, caracterul ***** (asterisc) este folosit și ca operator de redirectare (operator unar) și ca operator de multiplicare (operator aritmetic). Pentru a înmulți valorile memorate la două adrese de memorie indicate de doi pointeri p și q este obligatoriu să se folosească parantezele:

```
int a,b=10,c=20,*p=&b,*q=&c;
a=(*p)*(*q); /*variabilei a i se atribuie produsul dintre valoarea de la adresa memorată în p, adică valoarea variabilei b, și valoarea de la adresa memorată în q, adică valoarea variabilei c */
cout<<a; //afisează 200
```

Considerând declarația: `int a,b,*p=&a;` instrucțiunile de atribuire din tabelul alăturat sunt echivalente:

Așadar, **localizarea unei date memorate într-o variabilă de memorie** se poate face prin:

→ **adresare directă** – folosind numele variabilei de memorie (în exemplu a);

→ **adresare indirectă** – folosind o variabilă de tip pointer în care este memorată adresa zonei de memorie în care este stocată data (în exemplu, p).

<code>b=a+10;</code>	\leftrightarrow	<code>b=*&p+10;</code>
<code>a=b;</code>	\leftrightarrow	<code>*p=b;</code>
<code>a++;</code>	\leftrightarrow	<code>(*p)++;</code>
<code>a=a+5;</code>	\leftrightarrow	<code>*p=*&p+5;</code>
<code>b=a*a;</code>	\leftrightarrow	<code>b=(*p)*(*p);</code>
<code>a*=5;</code>	\leftrightarrow	<code>*p*=5;</code>

Temă



1. Scrieți un program în care declarați o variabilă de memorie de tip **float** și un pointer către tipul **float**, citiți de la tastatură valoarea variabilei și își afișați valoarea folosind cele două metode de adresare: adresarea directă și adresarea indirectă.
2. Scrieți un program în care declarați – o variabilă de memorie de tip înregistrare care va conține două câmpuri, corespunzătoare numitorului și numărătorului unei fracții, și o variabilă de tip pointer către tipul înregistrare –, citiți de la tastatură valorile câmpurilor din înregistrare și afișați valorile acestor câmpuri folosind cele două metode de adresare: adresarea directă și adresarea indirectă.

2.2.3.2. Operatorul de atribuire

Unei variabile de tip pointer i se poate atribui numai o valoare care reprezintă o adresă de memorie. Ea poate fi exprimată prin:

- O **adresă de memorie** obținută prin **aplicarea operatorului de adresare** pe o variabilă de memorie definită anterior, de același tip cu tipul de bază al pointerului. După operația de atribuire, variabila de tip pointer va adresa zona de memorie în care este stocată variabila de memorie.
- O altă **variabilă de tip pointer** care se referă la același tip de bază. După operația de atribuire, cele două variabile de tip pointer vor adresa aceeași zonă de memorie.
- O **constantă de tip adresă** de același tip cu tipul de bază al pointerului.
 - **Numele unui tablou de memorie** este o constantă de tip adresă. După operația de atribuire, variabila de tip pointer va adresa zona de memorie în care este stocat primul element al tabloului – în cazul vectorului, elementul cu indicele 0.
 - **Constanta 0 sau constanta simbolică predefinită NULL** – care corespunde pointerului zero – poate fi atribuită oricărui tip de pointer.
- Rezultatul unei expresii construite cu operanzi de tip adresă a unor date definite anterior și care aparțin tipului de bază al pointerului.

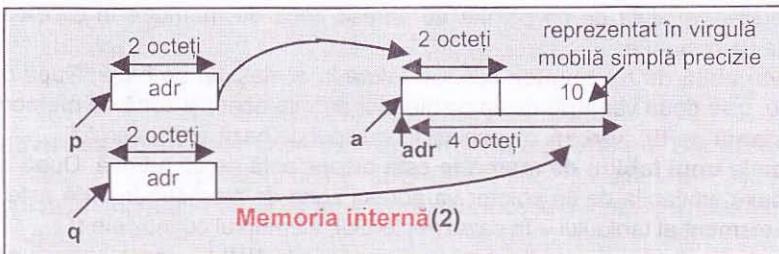
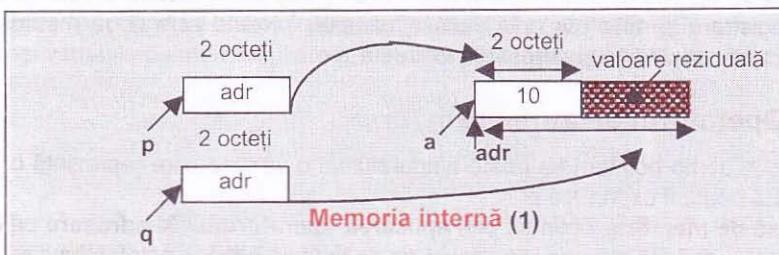
Observație. Unui pointer i se poate atribui un pointer de alt tip decât tipul de bază numai dacă se aplică **operatorul de conversie explicită de tip**. Acest operator trebuie aplicat însă cu foarte mare precauție, deoarece de multe ori rezultatul obținut nu este cel dorit.

```
int a=10, *p=&a; float *q;
q=(float*)p; //Memoria internă (1)
/*tipul expresiei (float*)p este float*, adică pointer către tipul float, și în acest mod pointerului q i s-a putut atribui adresa memorată în pointerul p*/
cout<<*q<<endl; // Afisează 1.401298e-44, un număr real.
```

```

/* Pointerul q indică o zonă de memorie de tip float de 4 octeți, din
care primii 2 octeți conțin reprezentarea în complement față de 2 a nu-
mărului întreg 10, iar următorii 2 octeți o valoare reziduală. Instruc-
țiunea de afișare face conversia valorii memorate, din reprezentarea în
format intern în reprezentarea în format extern, interpretând-o ca pe un
număr real reprezentat în virgulă mobilă simplă precizie */
*q=10; cout<<*q<<endl; // Afisează 10
p=(int*)q; // Memoria internă (2)
//pointerului p îi s-a atribuit adresa memorată în pointerul q
cout<<*p; // Afisează 0, un număr întreg.
/*Pointerul p indică o zonă de memorie de tip int de 2 octeți, care re-
prezintă primii 2 octeți din grupul de 4 octeți în care a fost reprezen-
tat în virgulă mobilă simplă precizie numărul real 10. Instrucțiunea de
afișare face conversia valorii memorate din reprezentarea în format
intern în reprezentarea în format extern, interpretând-o ca pe un număr
întreg reprezentat în complement față de 2 */

```



Exemplul 1:

```

char c='A',*p;
p=&c; /*variabilei p de tip pointer către caracter i se atribuie
        adresa variabilei de memorie c */
cout<<c<<" "<<p; //afisează A A

```

Exemplul 2:

```

int a=10,*p=&a,*q;
q=p; /* pointerului q îi se atribuie valoarea pointerului p, adică
        adresa variabilei a */
cout<<*p<<" "<<*q; //afisează 10 10

```

Exemplul 3: Numele unui vector este numele unei variabile de memorie, dar și o constantă de tip adresă cu tipul de bază al elementelor vectorului, care indică primul element al vectorului. Pointerului p îi se poate atribui adresa vectorului a exprimată prin:

- constanta simbolică a;
- operatorul de adresare & aplicat pe numele variabilei a;
- operatorul de adresare & aplicat pe identificatorul primului element al vectorului a[0], iar următoarele operații de atribuire sunt echivalente:

$p=a \leftrightarrow p=&a \leftrightarrow p=&a[0]$ și $*p \leftrightarrow a[0]$

```
int a[10], i, k=1, *p;
for (i=0; i<10; i++) a[i]=k++;
p=a;           /*variabilei p i se atribuie numele vectorului a, care
                  este o constantă simbolică de tip adresă */
cout<<*p<< " <<a[0]<<endl; //afișează 1 1 - valoarea elementului a[0]
```

Exemplul 4: Numele unei matrice este numele unei variabile de memorie, dar și o constantă de tip adresă cu tipul de bază al elementelor matricei, care indică primul element al matricei. Pentru un pointer p cu tipul de bază al elementelor matricei a, care trebuie să indice primul element al matricei, următoarele operații de atribuire sunt echivalente:

$p=a \leftrightarrow p=&a \leftrightarrow p=a[0] \leftrightarrow p=&a[0][0]$ și $*p \leftrightarrow a[0][0]$

```
int a[10][10], i, j, k=1, *p;
for (i=0; i<10; i++)
    for (j=0; j<10; j++) a[i][j]=k++;
p=a;           /*variabilei p i se atribuie numele matricei a, care
                  este o constantă simbolică de tip adresă */
cout<<*p<< " <<a[0][0]<<endl; //afișează 1 1 - valoarea elementului a[0][0]
```

Exemplul 5:

```
int a[4][5], i, j, k=1, *p;
for (i=0; i<4; i++)
    for (j=0; j<5; j++) a[i][j]=k++;
p=a[0];         /*variabilei p i se atribuie adresa vectorului format
                  din prima linie a matricei a, care este o constantă */
cout<<*p<< " <<a[0][0]<<endl;
//afișează 1 1 - valoarea elementului a[0][0]
```

Exemplul 6:

```
int *q;
q=0;           // pointerului q i se atribuie constanta 0
cout<<q<<endl; /*nu afișează nimic, deoarece pointerul q are
                  valoarea 0 și el nu indică nici o adresă de memorie */
q=NULL;         // pointerului q i se atribuie constantă simbolică NULL
cout<<q;         /*nu afișează nimic deoarece pointerul q are
                  valoarea NULL și el nu indică nici o adresă de memorie */
```



Unui pointer **nu i se poate atribui valoarea unei constante întregi**, chiar dacă este o constantă hexazecimală, deoarece programatorul nu are acces direct la adresele de memorie pentru a le gestiona. Unei date nu poate să îi atribue o adresă de memorie decât sistemul de operare.



Deoarece numele tabloului de memorie este folosit de compilatorul C++ ca o **constantă simbolică de tip adresă**, unei variabile de tip tablou nu i se poate atribui o altă variabilă de tip tablou sau un pointer, ca în exemplul următor:

```
int a[10], b[10], *p=&b;
a=b; a=p; // Eroare! Unei constante nu i se poate modifica valoarea
```

Temă

Scriți un program în care declarați o variabilă de memorie de tip înregistrare, care conține două câmpuri corespunzătoare numitorului și număratorului unei fracții și o variabilă de tip pointer către tipul înregistrare și care să realizeze următoarele:

- atribuie pointerului adresa variabilei înregistrare;
- citește de la tastatură valorile pentru numărător și numitor;
- afișează fracția simplificată – rezultatele se obțin în două variante, folosind câte una dintre cele două metode de adresare (adresarea directă și apoi adresarea indirectă) și se compară rezultatele obținute.

2.2.3.3. Operatorii aritmetici

Operațiile aritmetice permise asupra pointerilor sunt:

- Adunarea și scăderea unei constante – operatorii + și -.
- Incrementarea și decrementarea unui pointer – operatorii ++ și --.
- Scăderea a doi pointeri de același tip – operatorul -.

Atenție

Toate operațiile aritmetice cu pointeri către un tip de bază se execută considerând **unitatea egală** cu **sizeof(tip_bază)**, adică egal cu numărul de octeți necesari pentru a memora o dată care are tipul **tip_bază**. De exemplu, incrementarea unui pointer către tipul **int** va considera unitatea egală cu 2 (numărul de octeți necesari pentru reprezentarea tipului **int**) și noua adresă se va obține prin adunarea constantei 2 la adresa inițială memorată de pointer.

Operatorul pentru **adunarea** sau **scăderea unei constante** este:

$$p + k \text{ sau } p - k$$

unde **p** este un pointer către tipul **tip_baza** care memorează o adresă **adr**, iar **k** o constantă. Rezultatul este o adresă care se calculează astfel: **adr+k×sizeof(tip_baza)**, respectiv **adr-k×sizeof(tip_baza)**.

Observație: Operația de adunare este comutativă: **p+k = k+p**.

Operatorul pentru **incrementarea** și **decrementarea unui pointer** este:

$$p++ \text{ sau } p-$$

unde **p** este un pointer către tipul **tip_baza** care memorează o adresă **adr**. Rezultatul este o adresă care se calculează astfel: **adr+sizeof(tip_baza)**, respectiv **adr-sizeof(tip_baza)**.

Operatorul pentru **scăderea a doi pointeri** este:

$$p - q$$

unde **p** și **q** sunt doi pointeri către același tip de bază, care indică elementele aceluiași tablou de memorie, adresa memorată de **p** fiind mai mare decât cea memorată de **q** (**p** indică un element de tablou situat în memorie după elementul indicat de pointerul **q**). Rezultatul este un număr întreg care reprezintă numărul de elemente aflate între **p** și **q**.

Atenție

Operațiile aritmetice cu pointeri au sens numai dacă **adresele indicate de pointerii operanzi și de pointerul rezultatului expresiei se păstrează în spațiul de adrese ale unui tablou de memorie**. Sin-

gura excepție acceptată este cea a pointerului care indică adresa unui element situat imediat după ultimul element al tabloului.

Exemplul 1 – În cazul în care operatorii aritmetici se aplică pe pointeri care nu și păstrează valorile în spațiul unui tablou de memorie, rezultatele obținute nu au nici un fel de relevanță.

```
int a=10,*p=&a,*q=p; cout<<p<<" "<<*p<<endl;
p++; cout<<p<<" "<<*p<<endl; p++; cout<<p<<" "<<*p<<endl;
/*Zona de memorie indicată de pointer după fiecare operație de
incrementare conține o valoare reziduală care nu are nici un fel de
relevanță în cadrul programului*/
cout<<p-q; // afișează 2
```

Exemplul 2 – Se inversează un vector în el însuși. Se folosesc doi pointeri **p** și **q** către elementele vectorului care se interschimbă.

```
int a[20],n,*p,*q,aux;
cout<<"n= ";cin>>n;
for (p=a;p<a+n;p++) {cout<<"elementul "<<p-a+1<<"= "; cin>>*p;}
for(p=a,q=a+n-1;p<q;p++,q--) {aux=*p; *p=*q; *q=aux;}
for(p=a;p<a+n;p++) cout<<*p<<" ";
```

Observație. Prin definiție, operatorul **indice al tabloului ([])**:

- se aplică pe doi operanzi **x** și **y** (**x[y]** sau **y[x]**), **x** fiind un **pointer** și **y** un **întreg**;
- realizează adunarea dintre pointerul **x** și constanta **y**, obținând adresa elementului **y** al vectorului de la adresa **x**;
- rezultatul furnizat este valoarea indicată de pointerul obținut, adică valoarea elementului **y** al vectorului de la adresa **x**: **x[y]=y[x] =*(x+y)**.

Este un **operator binar** și are **prioritate maximă** și **asociativitatea de la stânga la dreapta**.

Elementul **i** al vectorului **a** poate fi identificat prin **a[i]**, deoarece operatorul **indice al tabloului ([])** execută suma dintre pointerul **a** (o adresă) și constanta numerică **i**. Prin aplicarea acestui operator se obține valoarea de la adresa elementului **i**. Deoarece operația de adunare dintre un pointer și o constantă este comutativă, următoarele expresii sunt echivalente și se pot folosi pentru identificarea elementului **i** al vectorului:

$$\mathbf{a[i]} \leftrightarrow \mathbf{* (a+i)} \leftrightarrow \mathbf{* (i+a)} \leftrightarrow \mathbf{i [a]}$$

Un element al matricei **a** poate fi identificat, folosind cele două indici **i** și **j**, prin **a[i][j]**. Asociativitatea operatorului **indice al tabloului** fiind de la stânga la dreapta și prioritatea sa fiind cea mai mare, calcularea adresei elementului se va face astfel: mai întâi se execută suma dintre pointerul **a** și constanta **i** și apoi suma dintre rezultatul obținut anterior și constanta **j**. Prin aplicarea operatorilor **indice al tabloului** se obține adresa elementului din linia **i** și coloana **j**, următoarele expresii fiind echivalente:

$$\mathbf{a[i][j]} \leftrightarrow \mathbf{* (a[i]+j)} \leftrightarrow \mathbf{(*(a+i)) [j]} \leftrightarrow \mathbf{* (* (a+i)+j)} \leftrightarrow \\ \mathbf{* (&a[0][0]+n*i+j)}$$

2.2.3.4. Operatorii relaționali

Operatorii relaționali care pot fi aplicati asupra pointerilor sunt:

- **Operatorii de inegalitate: <, >, <= și >=.**
- **Operatorii de egalitate: == și !=.**

Evaluarea unui operator relațional aplicat pe doi pointeri **p** și **q** se face prin analiza diferenței **p-q**.

Exemplu – Se afișează ordinea în care apar două valori **x** și **y** într-un vector cu numere întregi. Se folosesc doi pointeri **p** și **q** către elementele vectorului. Se parcurge vectorul folosind pointerul **p** pentru localizarea valorii **x** și pointerul **q** pentru localizarea valorii **y**. Stabilirea ordinii de apariție în vector se face prin compararea celor doi pointeri (a adreselor memorate în pointeri).

```
int x,y,a[100],*p=a,*q=a;
cout<<"n= ";cin>>n;
for (;p<a+n;p++) {cout<<"elementul "<<p-a+1<< "=" ; cin>>*p;}
cout<<"prima valoare= "; cin>>x;
cout<<"a doua valoare= "; cin>>y;
p=a;
while (*p!=x) p++;
while (*q!=y) q++;
if (p>q) cout<<"valoarea "<<y<<" apare inaintea valorii "<<x;
else cout<<"valoarea "<<x<<" apare inaintea valorii "<<y;
```



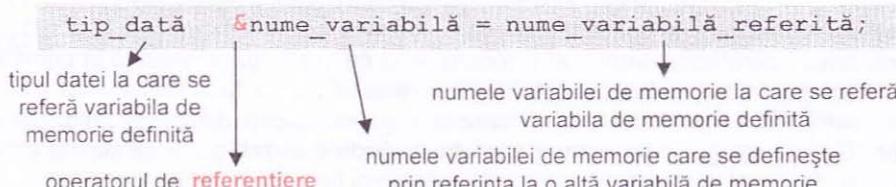
Tema

Scrieți următoarele programe, în care prelucrarea vectorilor se va face cu ajutorul pointerilor. Elementele vectorilor se citesc de la tastatură.

1. Se șterge elementul cu valoarea **k** dintr-un vector cu numere întregi. Valoarea numărului **k** se citește de la tastatură.
2. Se inserează elementul cu valoarea **x** înaintea elementului cu valoarea **y**. Valorile numerice **x** și **y** se citesc de la tastatură.
3. Se inserează elementul cu valoarea **x** după elementul cu valoarea **y**. Valorile numerice **x** și **y** se citesc de la tastatură.
4. Se ordonează crescător elementele vectorului.
5. Se afișează elementele comune din doi vectori **a** și **b** – o singură dată.

2.3. Tipul de date referință

Acest tip de dată permite folosirea mai multor identificatori pentru aceeași variabilă de memorie. Declararea unei variabile de tip referință se poate face prin instrucțiunea declarativă:



Exemplu:

```
int a=10;
int &b=a;
sau
int a=10,&b=a;
```

Variabila de memorie **b** este o referință către variabila de memorie **a**. Aceasta înseamnă că, prin referință, variabilei de memorie (**a**) i s-a mai dat un nume (**b**). Cele două variabile de memorie (**a** și **b**) au aceleași atribută (adresă de memorie, lungime, valoare, tip etc.), cu excepția numelui. Chiar dacă în program s-au declarat două variabile de memorie (**a** și **b**), se lucrează cu o singură zonă de memorie, care poate fi identificată cu două nume diferite, iar următoarele două instrucțiuni vor afișa aceleași valori:

```
cout<<"a = "<<a<<"adresa lui a= "<&a<<endl;
cout<<"b = "<<b<<"adresa lui b= "<&b;
```

Orice operație de modificare a valorii variabilei referite prin numele **b**, va modifica valoarea variabilei identificate prin numele **a**, deoarece ambele nume de variabile se referă la conținutul aceleiași zone de memorie. Următoarele două instrucțiuni vor afișa aceleași valori:

```
b=15;
cout<<"a = "<<a<<"adresa lui a= "<&a<<endl;
cout<<"b = "<<b<<"adresa lui b= "<&b;
```

Observații :

1. Tipul de dată referință, la fel ca și tipul de dată pointer, **conține o adresă**. Pentru a avea acces prin referință la o variabilă de memorie nu este însă nevoie să se folosească adresa de memorie, ci numele variabilei referință.

Exemplu:

```
int a=10,b=20; //s-au declarat două variabile întregi
int &x=a; //s-a declarat o referință la variabila a
int *p=&a; //s-a declarat un pointer care indică variabila a
int *q=&x; //s-a declarat un pointer care indică variabila x
cout<<a<<" " <<x<<" " <<*p<<" " <<*q<<" " <<endl;
//afișează 10 10 10 10
*p=20; cout<<a<<" " <<x<<" " <<*p<<" " <<*q<<" " <<endl;
//afișează 20 20 20 20
x=30; cout<<a<<" " <<x<<" " <<*p<<" " <<*q<<" " <<endl;
//afișează 30 30 30 30
p=&b; //p indică variabila b
cout<<b<<" " <<*p<<endl; //afișează 200 200
p=&x; //p indică variabila x care se referă la variabila a
cout<<b<<" " <<*p<<endl; //afișează 200 30
```

2. Operatorii aritmetici aplicați pe variabile referință nu modifică valoarea adresei referite, ci valoarea memorată la adresa variabilei referite. Operatorii relaționali aplicați pe variabile referință nu compară valorile adreselor referite, ci valorile memorate la adresele variabilelor referite.

Exemplu:

```
int a=10,b=20,&x=a,&y=b;
x=30; cout<<a<<" " <<x<<endl; //afișează 30 30
x++; cout<<a<<" " <<x<<endl; //afișează 31 31
x=--x; x+=x*2; cout<<a<<" " <<x<<endl; //afișează 90 90
if (x!=y) cout<<"adevarat" <<endl; //afișează adevarat
x=++y; cout<<a<<" " <<x<<endl; //afișează 21 21
cout<<b<<" " <<y<<endl; //afișează 21 21
if (x==y) cout<<"adevarat" <<endl; //afișează adevarat
```



1. Referința trebuie inițializată chiar în momentul declarării ei.

Exemplu:

```
int a,&b; // Eroare: variabila referință nu a fost inițializată
```

2. După ce a fost inițializată, referința nu mai poate fi modificată (nu se mai poate schimba adresa de memorie la care se referă).

Exemplu:

```
int a=10,b=100,&x=a;
cout<<a<<" "<<x<<endl; //afișează 10 10
x=&b; /* Eroare: variabilei referință x nu i se poate atribui
adresa variabilei de memorie b, deoarece variabila x, prin definiție,
se referă la variabila de memorie a și memorează adresa ei */
x=b; /* Corect: variabilei referință x i se poate atribui
valoarea variabilei de memorie b, aceasta însemnând că variabilei
referite de x (variabila a) i se atribuie valoarea variabilei b */
cout<<a<<" "<<x<<endl; //afișează 100 100
```

3. Nu se poate crea un pointer la o referință.

Exemplu:

```
int a=10,&x=a;
int *p=x; /* Eroare: pointerului p nu i se poate atribui adresa
memorată în variabila x, deoarece numele variabilei x nu identifică o
adresă, ci o valoare întreagă memorată la adresa variabilei a */
int *p=&x; /* Corect: pointerului p i se poate atribui adresa
variabilei referință x, aceasta fiind de fapt adresa variabilei a */
```

4. Nu este permisă declararea unei referințe la o adresă exprimată printr-o referință sau printr-un pointer.

Exemplu:

```
int a=10,b=100,*p=&a,*q=&b,&x=a;
int &y=&x; /* Eroare: s-a declarat variabila y care trebuie să se refere
la o variabilă de tip int, ca o referință la adresa variabilei x */
int &y=x; /* Corect: s-a declarat variabila y ca o referință la
variabila x, care este o referință la variabila a; variabila y se referă
la variabila a */
int &z=q; /* Eroare: s-a declarat variabila z, care trebuie să se refere
la o variabilă de tip int, ca o referință la variabila
pointer q care memorează o adresă */
int &z=*q; /* Corect: s-a declarat variabila z, care trebuie să se refere
la o variabilă de tip int, ca o referință la o variabilă
a cărei adresă este indicată de pointerul q (adică variabila b)*/
cout<<a<<" "<<*p<<" "<<x<<" "<<y<<endl; //afișează 10 10 10 10
cout<<b<<" "<<*q<<" "<<z<<endl; //afișează 100 100 100
*p=20; cout<<a<<" "<<*p<<" "<<x<<" "<<y<<endl;
//afișează 20 20 20 20
x=30; cout<<a<<" "<<*p<<" "<<x<<" "<<y<<endl;
//afișează 30 30 30 30
p=&z; b=200;
cout<<a<<" "<<*p<<" "<<x<<" "<<y<<" "<<*q<<" "<<z<<endl;
//afișează 30 200 30 30 200 200
```

5. Tipul referinței și tipul variabilei la care se referă trebuie să fie același și referința și obiectul la care face referirea trebuie să fie de același tip. Astfel, dacă referința

este o variabilă de memorie, referirea trebuie să se facă tot la o variabilă de memorie, și nu, de exemplu, la o constantă, chiar dacă aceasta este de același tip cu referința. În cazul în care nu se respectă aceste reguli, compilatorul vă va atenționa. Aceasta înseamnă că programul va putea fi executat, dar rezultatul obținut nu va fi cel așteptat, deoarece se creează o variabilă de memorie temporară, cu tipul referinței, căreia i se atribuie valoarea variabilei referite, respectiv a constantei, iar variabila referință va adresa această variabilă temporară, și nu variabila referită, respectiv constanta:

Exemplu:

```
char c='A'; float &a=c;
//echivalent cu: char c='A'; float temp=c; float &a=temp;
//temp fiind variabila de memorie temporară
cout<<a<<" "=><c<<endl; //afisează 65 A
a=a/5; cout<<a<<" "=><c<<endl; //afisează 13 A
c++; cout<<a<<" "=><c<<endl; //afisează 13 B
int &a=10; //echivalent cu: int temp=10; int &a=temp;
//temp fiind variabila de memorie temporară
cout<<a<<endl; //afisează 10
a=20; cout<<a; //afisează 20
```

Temă



Scrieți un program în care declarați o variabilă de memorie de tip **float** și o referință către această variabilă de memorie, citiți de la tastatură și afișați valoarea variabilei de tip **float** – folosind cele două nume ale variabilei de memorie.

Utilitatea tipului de date referință

Tipul de date referință este util în transmiterea datelor între modulul apelant și subprogram prin intermediul parametrilor de comunicație. În timpul execuției subprogramului, datele cu care el lucrează (variabilele locale și parametrii cu care a fost apelat) sunt păstrate în stivă. Instrucțiunile subprogramului pot modifica aceste date. Modificările se execută asupra valorilor memorate în stivă. Când se termină execuția subprogramului, spațiul ocupat în stivă de parametri și de variabilele locale este eliberat și – în cazul parametrilor de ieșire sau de intrare-ieșire – se pierd valorile calculate în subprogram.

```
void sb(int a) {a=a+2; cout<<a<<endl;;} //afisează 4
void main() {int a=2; sb(a); cout<<a; } //afisează 2
```

Folosind **transferul parametrului prin referință**, în momentul apelării subprogramului, în stivă este încărcată adresa de memorie la care se găsește valoarea parametrului. Subprogramul va lucra direct în zona de memorie în care se găsește data. Așadar, atât modulul apelant cât și subprogramul lucrează asupra aceleiași date, și orice modificare a valorii acestui parametru făcută în subprogram se va reflecta și în modulul apelant. La terminarea execuției subprogramului, este eliberată din stivă zona în care este memorată adresa parametrului.

```
void sb(int &a) {a=a+2; cout<<a<<endl;;} //afisează 4
void main() {int a=2; sb(a); cout<<a; } //afisează 4
```

Între modulul apelant și subprogram se pot transmite și **parametri de tip adresă**. Dacă adresa este transmisă prin valoare și ea este modificată în subprogram, noua valoare se pierde la terminarea execuției subprogramului.

```
void sb(int *p) {int x=4; p=&x cout<<p<<endl;} //afisează 4
void main() {int a=2,*p=&a; sb(p); cout<<p; } //afisează 2
```

Folosind **transferul parametrului prin referință** pentru pointer, în momentul apelării subprogramului, în stivă este încărcată adresa de memorie a pointerului și subprogramul va lucra direct în zona de memorie în care se găsește pointerul. Așadar, atât modulul apelant cât și subprogramul lucrează asupra aceleiași date, și orice modificare a adresei memorate în pointer făcută în subprogram se va reflecta și în modulul apelant. La terminarea execuției subprogramului, este eliberată din stivă zona în care este memorată adresa pointerului.

```
void sb(int *p) {int x=4; p=&x; cout<<*p<<endl;} //afișează 4
void main() {int a=2,*p=&a; sb(p); cout<<*p;} //afișează 4
```

2.4. Alocarea dinamică a memoriei

Declarările de date în cadrul programului folosesc **alocarea statică a memoriei**.

Alocarea statică a memoriei se face în timpul **compilării**, în funcție de modul în care a fost declarată variabila de memorie sau structura de date, iar în timpul execuției programului nu mai poate fi modificată.

Variabilele care folosesc alocarea statică a memoriei se numesc **variabile statice**. Sistemul de operare le alocă spațiu în segmentul de date sau în stiva sistemului, în funcție de modul în care au fost declarate în program (globale sau locale).

Structurile de date statice au un mare dezavantaj, deoarece în timpul execuției programului pot să apară următoarele cazuri:

- spațiul alocat structurii este **insuficient** și este posibil ca să se depășească acest spațiu și să se între în spațiul de memorie alocat altor date;
- spațiul alocat structurii este **mult mai mare decât este necesar** și memoria internă nu este utilizată eficient.

Acest dezavantaj poate fi eliminat prin folosirea alocării dinamice a memoriei.

Alocarea dinamică este metoda prin care unei variabile de memorie sau unei structuri de date i se atribuie spațiu de memorie – sau se eliberează spațiul de memorie ocupat de ea în timpul **execuției** programului, în funcție de necesități.

Variabilele care folosesc alocarea dinamică a memoriei se numesc **variabile dinamice**. Spațiul de memorie alocat va fi în **zona de adrese libere (Heap)**. Sistemul de operare aloca unei date zonă de memorie în Heap sau eliberează spațiul de memorie alocat datei, în urma unor **cereri de alocare de memorie și de eliberare de memorie** din program. Mecanismul prin care programatorul folosește alocarea dinamică a memoriei este următorul:

1. Se declară o variabilă de memorie de tip pointer către tipul de dată al variabilei dinamice.

```
<tip_de_baza> *<nume_pointer>;
```

La compilarea programului, variabilei pointer i se va aloca spațiu în segmentul de date sau în stiva sistemului, în funcție de modul în care a fost declarat în program (global sau local). În variabila pointer se va memora adresa variabilei dinamice.

2. În momentul în care în program trebuie folosită variabila dinamică se cere sistemului de operare să îi aloce spațiu în Heap. Pentru cererea de alocare de memorie se folosește operatorul **new**:

```
<nume_pointer> = new <tip_de_baza>;
```

Sistemul de operare caută în Heap o zonă de memorie liberă, de dimensiunea tipului variabilei dinamice, și atribuie pointerului adresa acestei zone.

3. În momentul în care în program nu mai este folosită variabila dinamică se cere sistemului de operare să elibereze spațiul de memorie pe care îl-a alocat în Heap. Pentru cererea de eliberare a memoriei se folosește operatorul **delete**:

```
delete <nume_pointer>;
```

Sistemul de operare declară liberă zona de memorie care a fost alocată variabilei dinamice, și consideră că variabila pointer nu mai este inițializată cu o adresă de memorie, iar valoarea care a fost memorată în variabila dinamică se pierde.

Observație. Operatorii **new** și **delete** sunt operatori **unari** și au **prioritatea** și **asociativitatea** acestui tip de operatori.

Studiu de caz

Scop: exemplificarea modului în care puteți folosi alocarea dinamică a memoriei.

Enunțul problemei 1: Se citesc de la tastatură trei numere întregi – **a**, **b** și **c** – care reprezintă mărimile laturilor unui triunghi. Să se afișeze aria triunghiului.

Se folosește **alocarea dinamică a variabilelor elementare**. Pentru referirea mărimilor laturilor se folosesc pointerii **a**, **b** și **c** către tipul **int**, iar pentru referirea ariei și a semi-perimetrului se folosesc pointerii **aria** și **sp** către tipul **float**.

```
#include<iostream.h>
#include<math.h>
void main()
{int *a,*b,*c; float *aria, *sp; //se declară pointerii (1)
 a = new int; b = new int; c = new int; //se alocă memorie pentru
 aria = new float; sp = new float; //variabilele dinamice (2)
 //se citesc de la tastatură valorile variabilelor dinamice
 //folosite pentru laturile triunghiului
 cout<<"a = "; cin>>*a; cout<<"b = "; cin>>*b; cout<<"c = "; cin>>*c;
 //se calculează valorile variabilelor dinamice folosite pentru
 //semiperimetru și arie
 *sp=(*a+*b+*c)/2.; *aria=sqrt(*sp*(*sp-*a)*(*sp-*b)*(*sp-*c));
 //se afișează valoarea variabilei dinamice folosite pentru arie
 cout<<"aria= "<<*aria;
 //se eliberează zona de memorie alocată variabilelor dinamice (3)
 delete a; delete b; delete c; delete aria; delete sp;}
```

Enunțul problemei 2: Se citesc de la tastatură coordonatele a două puncte din plan – **a** și **b**. Să se afișeze distanța dintre cele două puncte.

Se folosește **alocarea dinamică a structurii de date de tip înregistrare**. Pentru referirea coordonatelor celor două puncte se folosesc pointerii **a** și **b** către tipul **înregistrare punct**, iar pentru referirea distanței se folosește pointerul **d** către tipul **float**.

```
#include<iostream.h>
#include<math.h>
struct punct {int x,y;};
punct *a,*b; float *d; //se declară pointerii (1)
void main()
{//se alocă memorie pentru variabilele dinamice (2)
 a = new punct; b = new punct; d = new float;
```

```
//se citesc de la tastatură valorile variabilelor dinamice
//folosite pentru coördonatele punctelor a și b
cout<<"coördonatele punctului a - x: "; cin>>a->x;
cout<<"                                     - y: "; cin>>a->y;
cout<<"coördonatele punctului b - x: "; cin>>b->x;
cout<<"                                     - y: "; cin>>b->y;
//se calculează valoarea variabilei dinamice folosite pentru distanță
*d=sqrt(pow(a->x-b->x,2)+pow(a->y-b->y,2));
//se afișează valoarea variabilei dinamice folosite pentru distanță
cout<<"distanța dintre punctele a și b este "<<*d;
//se eliberează zona de memorie alocată variabilelor dinamice (3)
delete a; delete b; delete d;
```

Enunțul problemei 3: Se citesc de la tastatură **n** numere întregi care se memorează într-un vector. Să se inverseze vectorul în el însuși și să se afișeze vectorul după inversare.

Se folosește **alocarea dinamică a structurii de date de tip vector**. Pentru referirea numărului de elemente ale vectorului se folosește pointerul **n** către tipul **int**, iar pentru referirea variabilei prin intermediul căreia se interschimbă două elemente ale vectorului se folosește pointerul **aux** către tipul **int**. Pentru referirea vectorului se folosește pointerul **p** către tipul **int[n]**. Acest pointer indică o zonă continuă de **n*sizeof(int)** octeți, **n** reprezentând valoarea citită de la tastatură pentru numărul de elemente ale vectorului și care este referită de pointerul **n**. Pointerii **q** și **r** către tipul **int** se folosesc pentru a parcurge zona de memorie alocată vectorului în vederea prelucrării elementelor lui.

```
#include<iostream.h>
#include<math.h>
void main()
{int *p,*q,*r,*n,*aux;      //se declară pointerii (1)
 //se alocă memorie pentru variabilele dinamice (2)
 n=new int; aux=new int; cout<<"n= "; cin>>*n;
 p = new int[*n];
 //se citesc de la tastatură valorile pentru elementele vectorului
 for (q=p;q<p+*n;q++)
 {cout<<"elementul= "<<q-p+1<< "= "; cin>>*q;}
 //se interschimbă elementele vectorului
 for (q=p,r=p+*n-1;q<r;q++,r--) {*aux=*q; *q=*r; *r=*aux;}
 //se afișează elementele vectorului
 for (q=p;q<p+*n;q++) cout<<*q<< " ";
 //se eliberează zona de memorie alocată variabilelor dinamice (3)
 delete []p; delete n; delete aux;}
```



Temă



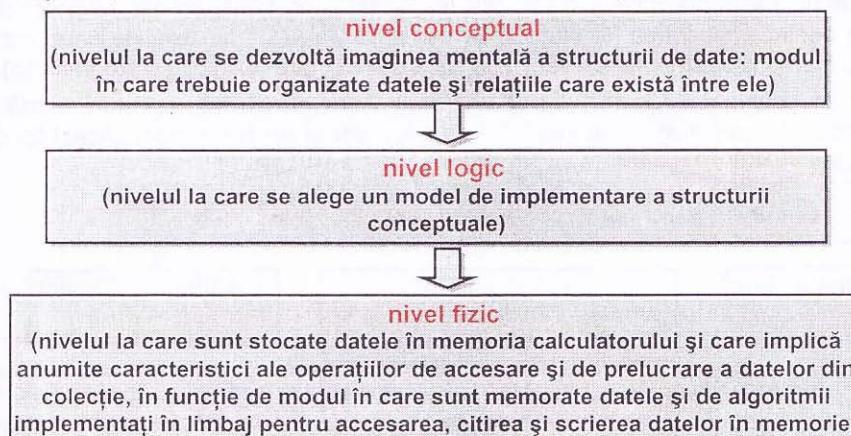
Scripti următoarele programe în care folosiți alocare dinamică a memoriei:
a. Citiți de la tastatură temperatura, presiunea și numărul de moli ai unui gaz și calculați volumul lui.

- Citiți de la tastatură număratorul și numitorul a două fracții, calculați suma și produsul celor două fracții, simplificați rezultatele obținute și apoi afișați suma și produsul.
- Citiți de la tastatură coordonatele a două colțuri ale unui dreptunghi (colțul din stânga sus și colțul din dreapta jos) și afișați dimensiunea diagonalei dreptunghiului.

- d. Citiți de la tastatură temperaturile medii zilnice din luna precedentă și afișați temperatura medie a lunii, și zilele în care temperaturile au fost mai mari decât media.

2.5. Clasificarea structurilor de date

Organizarea în structuri de date a datelor prelucrate de algoritmi simplifică multe dintre operațiile de prelucrare. Atunci când organizați datele într-o structură de date, trebuie să identificați modul în care puteți avea acces la ele și operațiile pe care le puteți executa cu datele din colecție. Procesul de organizare a datelor în colecții este un proces care se desfășoară pe trei niveluri care interacționează între ele, pornind de la nivelul conceptual:



Ați studiat structurile de date **implementate la nivel fizic** în limbajul C++:

- **tabloul de memorie** – structură de date omogenă, liniară, internă și temporară;
- **fișierul** – structură de date omogenă, liniară, externă și permanentă;
- **șirul de caractere** – structură de date omogenă, liniară, internă și temporară;
- **înregistrarea** – structură de date neomogenă, internă și temporară.

Ați mai studiat și o structură de date **logică lista** – structură de date omogenă, liniară, internă și temporară și o metodă de implementare la **nivel logic** a listei folosind vectorul.

Studiu de caz

Scop: exemplificarea modului în care identificați structura de date pe care o folosiți pentru a rezolva problema.

Enunțul problemei 1. O firmă de transport are un parc de 10 mașini cu capacitatea de transport diferite. Trebuie să se determine câte dintre aceste mașini au cea mai mare capacitate de transport.

Pentru rezolvarea problemei trebuie stabilită structura de date care se va folosi:

- La **nivel conceptual** – capacitatea de transport ale mașinilor reprezintă un sir de numere întregi, aranjate într-o ordine aleatorie, în care trebuie căutat numărul cel mai mare și de câte ori apare în sir.

20	40	50	30	20	40	50	40	30	40
----	----	----	----	----	----	----	----	----	----

- La **nivel logic** – implementarea permisă de limbajul C++ a unei colecții de date omogene este vectorul, fiecare număr întreg fiind un element al structurii. Pentru rezolvarea problemei 1.

mei se vor folosi următorii algoritmi: algoritmul pentru parcurgerea vectorului la memorarea numerelor, algoritmul pentru determinarea valorii maxime dintr-un sir de numere și algoritmul de căutare în vector a elementelor cu o valoare precizată (valoarea maximă).

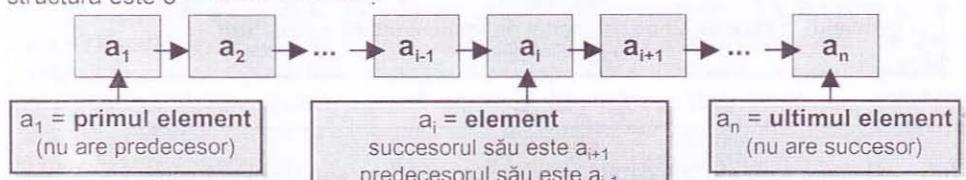
```
int a[10];
```

- La nivel fizic – numerele vor fi memorate într-o zonă continuă de memorie internă, fiecărui număr alocându-i-se același spațiu pentru memorare. Identificarea unui element al structurii se face prin numărul său de ordine (indicele).

20	40	50	30	20	40	50	40	30	40
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Dacă se dorește păstrarea datelor memorate (capacitățile de transport ale mașinilor din parc auto) pentru prelucrarea lor ulterior, se vor salva într-un fișier text, de unde vor fi restaurate în memoria internă într-un vector, la fiecare prelucrare (execuție a programului).

Observație. În structura de date folosită (vectorul), între elementele structurii există o relație de ordine în care fiecare element are un succesor și un predecesor. Acest tip de structură este o **structură liniară**.



Enunțul problemei 2. Un profesor trebuie să calculeze media semestrială a fiecărui elev din clasă și să obțină o listă cu numele, prenumele și mediile elevilor, în ordinea descrescătoare a mediilor.

Pentru rezolvarea problemei, trebuie stabilită structura de date care se va folosi:

- La nivel conceptual – elevii din clasă reprezintă un sir de entități care sunt caracterizate de o listă de proprietăți (attribute): nume și prenume (care identifică unic elevul), cinci note și media semestrială.

	Nume	Prenume	Nota 1	Nota 2	Nota 3	Nota 4	Nota 5	Media
1	Anghel	Maria	10	9	10	9		9,5
...
30	Vlad	Mihai	7	6	8			7

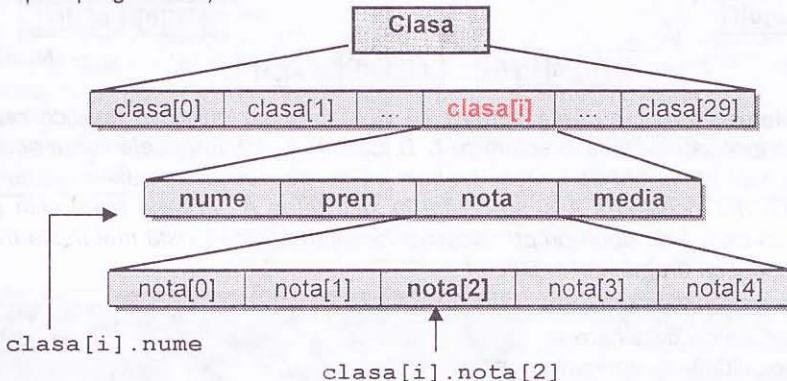
- La nivel logic – implementarea permisă de limbajul C++ a unei colecții de date omogene este vectorul, fiecare listă de attribute ale unui elev fiind un element al structurii. Pentru implementarea listei de attribute (care este formată din date neomogene: numele și prenumele sunt siruri de caractere, notele sunt numere întregi, iar media este un număr real) se va folosi structura de date de tip înregistrare, fiecare atribut fiind memorat într-un câmp. Pentru rezolvarea problemei se vor folosi următorii algoritmi: algoritmul pentru parcurgerea vectorului la memorarea listei de attribute a fiecărui elev și la afișarea mediilor, algoritmul pentru calcularea mediei aritmetice și algoritmul de sortare a elementelor vectorului în funcție de valoarea unui atribut din lista de attribute.

```
struct elev
{
    char nume[20], pren[20];
    unsigned nota[5];
    float media;
};

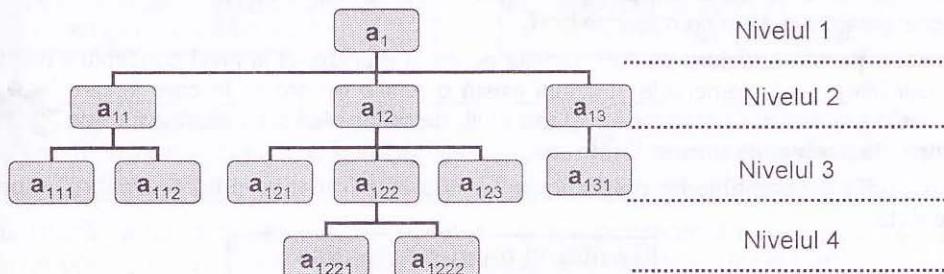
elev clasa[30];
```

→ La nivel **fizic** – listele de atribută ale fiecărui elev vor fi memorate într-o zonă continuă de memorie internă, fiecărei liste de atribută alocându-i-se același spațiu pentru memorare. Identificarea unei liste de atribută în cadrul structurii se face prin numărul său de ordine (indicele), iar identificarea unui atribut din listă se face prin numele câmpului.

Și în cazul acestui exemplu, dacă se dorește păstrarea datelor memorate (lista de atribută a fiecărui elev din clasă), pentru prelucrarea lor ulterior, se vor salva într-un fișier text, de unde vor fi restaurate în memoria internă într-un vector de înregistrări, la fiecare prelucrare (execuție a programului).



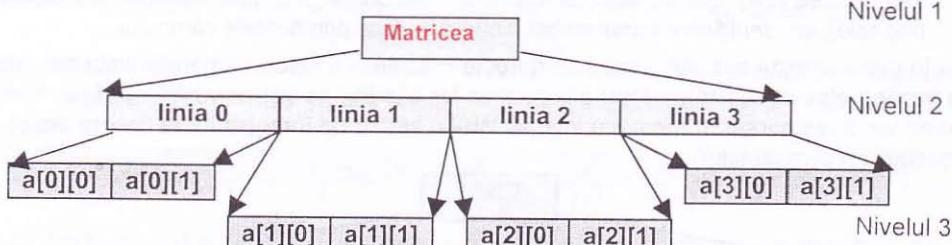
Observație. În structura de date folosită pentru lista de atribută, între elementele structurii există o relație de ordine în care fiecare element are un singur predecesor și nici unul, unul sau mai mulți succesiști. Entitatea *elev* (coresponde unui element din vectorul *clasa*) are ca predecesor entitatea *clasa* și ca succesiști entitățile: *nume*, *pren*, *nota* și *media*. Entitatea *nume* are un singur predecesor (entitatea *elev*) și nici un succesor. Entitatea *nota* are un singur predecesor (entitatea *elev*) și cinci succesiști (*nota[0]*, *nota[1]*, *nota[2]*, *nota[3]* și *nota[4]*). Structurile de date de tip înregistrare pot fi ierarhizate pe unul sau mai multe niveluri. Acest model de reprezentare a datelor se numește **arbore cu rădăcină**, iar acest tip de structură de date este o **structură ierarhizată**.



Într-o structură de date ierarhizată, datele pot fi grupate pe mai multe niveluri. Elementul *a₁* se găsește pe nivelul 1. El nu are predecesor, dar are trei succesiști: *a₁₁*, *a₁₂* și *a₁₃*, care se găsesc pe nivelul 2. Elementul *a₁₂* are un singur predecesor (*a₁*) și trei succesiști: *a₁₂₁*, *a₁₂₂* și *a₁₂₃*. Elementele *a₁₁₁*, *a₁₁₂*, *a₁₂₁*, *a₁₂₃* și *a₁₃₁* de pe nivelul 3 și elementele *a₁₂₂₁* și *a₁₂₂₂* de pe nivelul 4 nu au succesiști.

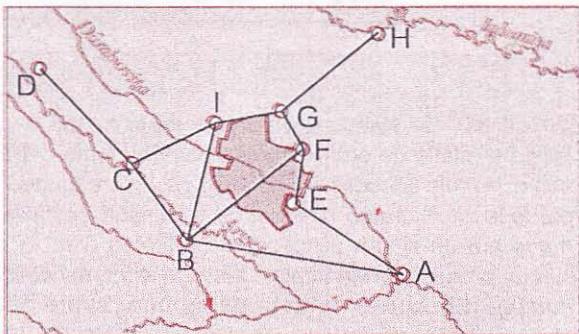
O matrice cu *n* linii și *m* coloane este și ea o structură de date ierarhizată pe trei niveluri: pe primul nivel este matricea care are *n* succesiști: liniile, care sunt pe nivelul 1. Fiecare

linie are ca predecesor matricea și ca succesișor cele m elemente de pe o linie, care se găsesc pe nivelul 3.



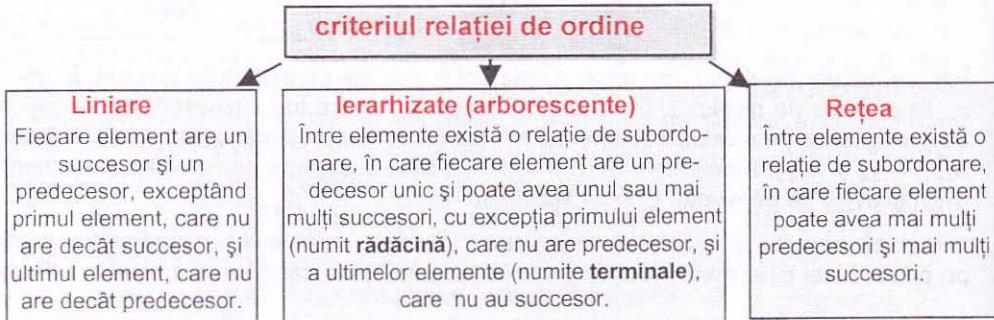
Enunțul problemei 3. O persoană dorește să viziteze șapte obiective turistice care se găsesc în șapte localități, pe care le notăm cu A, B, C, E, F, G și I. Între cele șapte obiective turistice există mai multe căi de acces. Trebuie să se găsească un traseu, astfel încât turistul să plece din localitatea A și să revină în localitatea A, vizitând toate cele șapte obiective fără să treacă de două ori prin aceeași localitate. Dacă există mai multe trasee, să se caute traseul cu drumul cel mai scurt.

Pentru rezolvarea problemei, trebuie stabilită structura de date care se va folosi. Localitățile reprezintă entități care pot fi legate sau nu prin căi de acces. Fiecare cale de acces este caracterizată de distanța dintre cele două localități. Pentru a reprezenta la nivel conceptual această structură de date vom reprezenta în plan harta zonei turistice prin intermediul unor elemente geometrice, astfel: localitățile se reprezintă prin cercuri în interiorul căroră vom scrie identificatorul localității, iar căile de acces prin linii drepte care unesc anumite puncte. Acest model de reprezentare a datelor se numește **graf**.



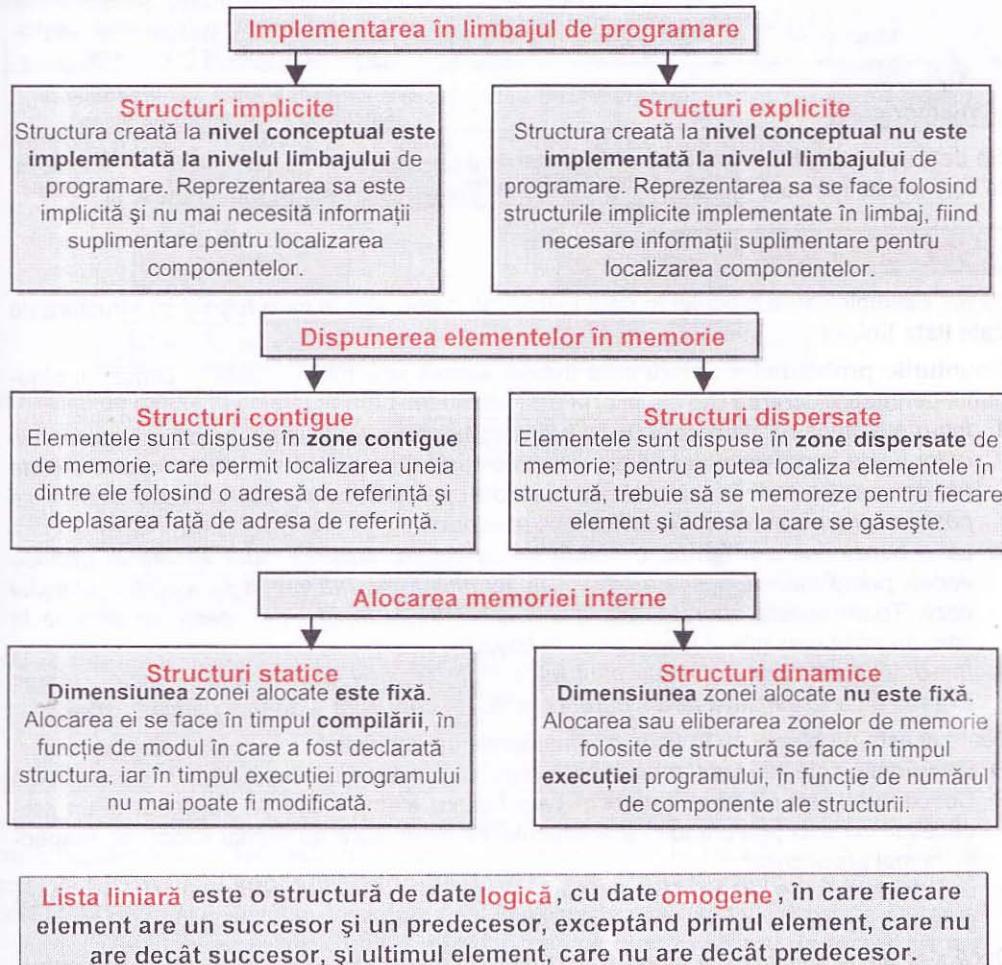
Observație. În structura de date folosită pentru a reprezenta la nivel conceptual entitățile (localitățile), între elementele structurii există o relație de ordine în care fiecare element are mai mulți predecesori și mai mulți succesișori. Acest tip de structură de date este o **rețea**.

Din studiul de caz anterior putem folosi un nou criteriu pentru **clasificarea structurilor de date**:



2.6. Lista liniară

Pe lângă criteriile studiate pentru clasificarea structurilor de date, mai există și următoarele criterii:



Lista liniară, la fel ca și vectorul, din punct de vedere conceptual este o **structură de date omogenă liniară**. Între cele două structuri există următoarele deosebiri:

Criteriu	Vectori	Liste
Implementarea în limbaj	Sunt structuri implicate . Fiind o structură fizică , este implementată în limbajul de programare. Nu necesită informații suplimentare pentru localizarea elementelor structurii în memoria internă, deoarece mecanismul prin care este implementată fizic asigură identificarea elementelor.	Sunt structuri explicite . Fiind o structură logică , trebuie aleasă o metodă de implementare folosind structurile fizice existente. Necesită informații suplimentare pentru localizarea elementelor structurii în memoria internă, deoarece mecanismul prin care este implementată fizic nu asigură identificarea elementelor.

Criteriu	Vectori	Liste
Dispunerea elementelor în memorie	Sunt structuri contigue . Predecesorul, elementul și succesorul se găsesc în locații contigute prin mecanismul de implementare a structurii de date la nivel fizic.	Pot fi structuri dispersate . Predecesorul, elementul și succesorul nu sunt în locații de memorie contigute. Programatorul trebuie să definească mecanismul prin care elementele structurii vor fi legate unele de altele pentru a putea fi regăsite în memorie.
Alocarea memoriei	Sunt structuri statici .	Sunt structuri statici sau dinamice , în funcție de implementarea aleasă.

Se definește **lungimea listei** (n) ca fiind numărul de elemente ale listei. **Lista vidă** este lista care are lungimea 0 (nu are nici un element). Elementele listei se mai numesc și **noduri**.

Studiu de caz

Scop: exemplificarea modului în care identificați o aplicație în care folosiți ca structură de date lista liniară.

Enunțurile problemelor pentru care trebuie aleasă structura de date și conceput algoritmul pentru prelucrarea lor:

1. Într-o bibliotecă există o colecție de cărți organizate în ordinea alfabetică a autorilor. Un cititor poate împrumuta o carte (se extrage o carte din colecție) sau poate înapoia o carte (se inserează o carte în colecție). Toate aceste operații trebuie executate astfel încât să se păstreze organizarea în ordine alfabetică a autorilor.
2. La o benzinărie s-a format o coadă de așteptare. Mașinile sunt servite în ordinea venirii: prima mașină sosită este servită, iar ultima mașină venită se așază la sfârșitul cozii. Toate aceste operații trebuie executate astfel încât să se păstreze ordinea în care au sosit mașinile și să se așeză la coadă.
3. Într-o stivă de cărți, volumele sunt așezate în ordinea alfabetică a titlurilor. Trebuie extrasă o carte din stivă fără a deranja modul în care sunt ordonate cărțile în stivă.

Toate aceste probleme au în comun următoarele caracteristici:

- Elementele colecției sunt **omogene** (înregistrări cu aceeași structură).
- Conceptual sunt structuri **liniare** în care fiecare element are un succesor și un predecesor, cu excepția primului și a ultimului element, care au numai succesor, respectiv numai predecesor.
- Structurile de date se modifică folosind aceiași **algoritmi de prelucrare** se inserează și se extrag elemente, păstrându-se o anumită ordine de aranjare a elementelor.

Dacă s-ar alege soluția de a grupa aceste elemente într-o structură de date de tip **vector**, algoritmii de prelucrare (de actualizare a vectorilor) vor necesita multe deplasări de elemente care consumă timp de prelucrare. Caracteristicile operațiilor de prelucrare a vectorilor sunt:

- Vectorul este o structură de date care are lungimea fizică fixă, iar implementarea lui la nivel fizic este un proces prin care se face conversia locației conceptuale a unui element, în locația reală, din memoria internă.
- Orice operație de adăugare sau extragere a unui element din colecție modifică lungimea logică a vectorului.
- Pentru a insera un element în colecție, trebuie deplasate toate elementele spre dreapta, începând cu poziția inserării.
- Pentru a extrage un element din colecție, trebuie deplasate toate elementele spre stânga, de la sfârșit, până în poziția din care se extrage.

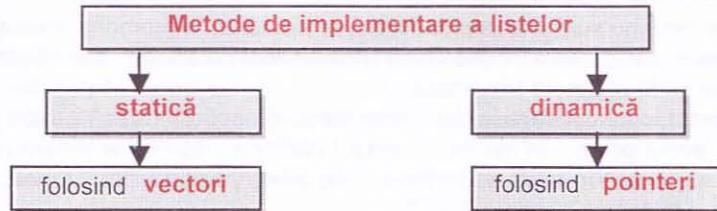
În cazul **structurilor liniare care trebuie să-și păstreze în timpul exploatarii ordonarea după un anumit criteriu**, mecanismul vectorilor este greoi. În aceste cazuri, se poate alege ca soluție de implementare a structurii de date **lista liniară** ce degrevează programatorul de ordonarea după indice a structurii, impusă de vectori.

Implementarea structurilor de date cu ajutorul listelor are următoarele **avantaje**:

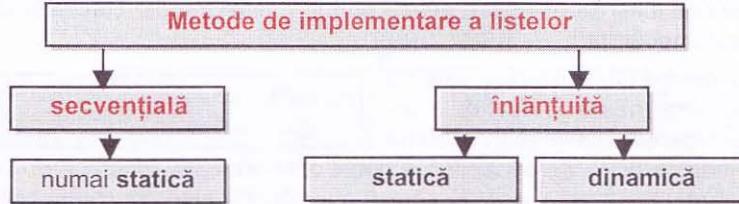
- alocarea dinamică a memoriei – care gestionează memoria mult mai eficient;
- algoritmii de prelucrare – care sunt mult mai eficienți (se execută mult mai puține operații pentru inserarea și ștergerea unui element).



În funcție de modul în care se alocă memoria internă, se pot folosi metodele:



În funcție de modul în care sunt aranjate elementele în listă, se pot folosi metodele:



Metoda de implementare	Avantajul alocării dinamice a memoriei	Avantajul algoritmilor de prelucrare
statică secvențială	nu	nu
- statică înlănțuită	nu	da
dinamică înlănțuită	da	da

Implementarea statică secvențială nu aduce niciunul dintre avantajele listelor, fiind o implementare în care lista este prelucrată la fel ca un vector. În **implementarea înlănțuită** nodurile listei **nu mai sunt stocate succesiv** în memorie. Această implementare se poate face atât static, cât și dinamic, între cele două implementări existând următoarele diferențe:

- în implementarea **statică**, nodurile listei ocupă un **bloc contiguu de locații de memorie** (zona de memorie alocată vectorului);
- în implementarea **dinamică**, nodurile listei ocupă **locații dispersate din memorie** (a căror adresă poate fi păstrată cu ajutorul **pointerilor**).

În clasa a X-a ați învățat modul în care puteți implementa static liste liniare. Algoritmii pentru prelucrarea unei liste înlănțuite sunt aceiași, atât în cazul implementării statice, cât și în cazul implementării dinamice.

2.6.1. Implementarea dinamică a listelor în limbajul C++

Exemplu – O listă este formată din 5 cuvinte (noduri), fiecare cuvânt având maxim 4 caractere. Nodurile listei se exploatează în ordine alfabetică:

Lista = {"alfa", "beta", "gama", "teta", "zeta"}

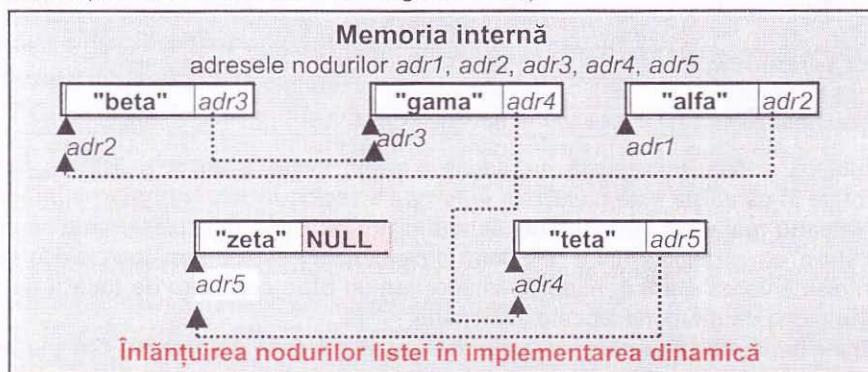


Deoarece nodurile listei nu sunt aranjate succesiv, ci aleatoriu, în memorie, trebuie implementat un mecanism prin care să se precizeze ordinea reală a acestor noduri (ordinea în care se înlanțuie în listă). Aceasta înseamnă că:

- trebuie cunoscut locul din care începe lista (lanțul de noduri), adică poziția primului nod din listă (nodul **prim**) – în exemplu, nodul "alfa".
- trebuie cunoscut locul în care se termină lista, adică poziția ultimului nod din listă (nodul **ultim**) – în exemplu, nodul "zeta".
- pentru fiecare nod din listă, cu excepția ultimului nod, trebuie cunoscut nodul care este succesorul lui – de exemplu, pentru nodul "gama" trebuie cunoscut că succesorul său este nodul "teta".

Metoda folosită pentru implementare este ca un nod al listei să conțină două tipuri de informații: **informația propriu-zisă** și **informația de legătură** – adresa prin care se identifică nodul care urmează în structură. Informația de legătură memorată în ultimul nod (în câmpul de adresă) este constanta **NUL** (care semnifică faptul că ultimul nod nu se leagă de nimic).

Informația propriu-zisă	Informația pentru legătură
-------------------------	----------------------------



Lista liniară înlanțuită este o structură logică de date, parcursă liniar, care are două extremități (**început** și **sfârșit**), în care fiecare element i se asociază o **informație suplimentară** referitoare la locul elementului următor, din punct de vedere logic.

Un nod al listei va fi de tipul **înregistrare** ce conține un câmp cu informația pentru legătură care este **adresa succesorului** exprimată printr-un pointer.

struct nod

```
{<tip_1> <info_11>, <info_12>, ..., <info_1n>;
 <tip_2> <info_21>, <info_22>, ..., <info_2n>;
 ...
```

```

<tip_m> <info_m1>, <info_m2>, ..., <info_mn>;
    nod *urm;};
nod *prim, *ultim, *p;

```

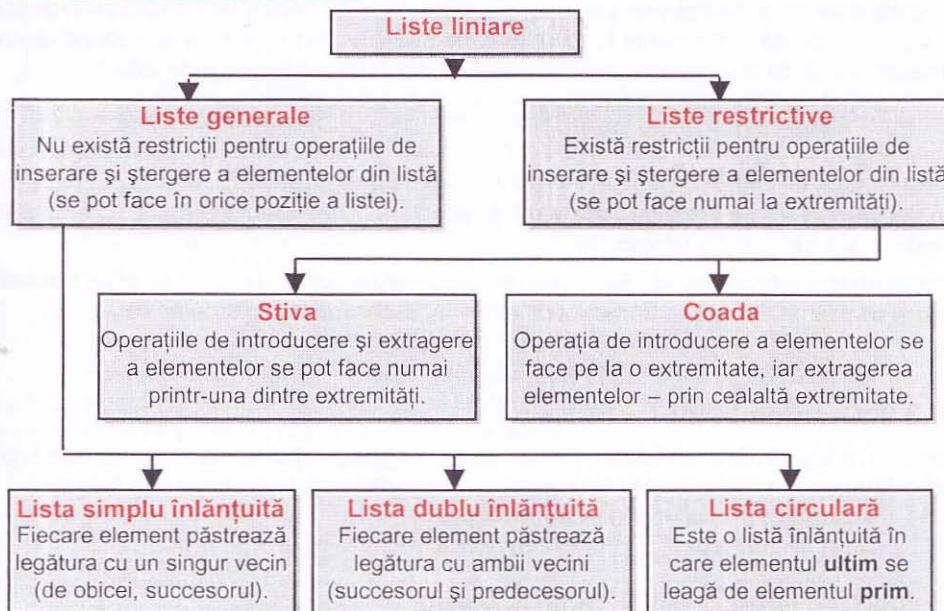
Câmpurile `<info_ij>` sunt câmpurile cu informații, iar câmpul `urm` este un câmp de tip pointer către tipul de bază `nod` și conține informația de legătură (adresa următorului nod din listă). Acest tip de înregistrare se numește **înregistrarea autoreferită**. S-au declarat variabilele de tip adresă a unei înregistrări de tip `nod`:

- `*prim` și `*ultim` pentru a memora adresa primului nod, respectiv a ultimului nod; ele vă ajută să identificați extremitățile listei;
- `*p` pentru a memora adresa unui nod curent din listă (este folosit pentru parcurgerea listei).

În liste, algoritmii de inserare și eliminare a unor noduri din structură se simplifică foarte mult:

- **Inserarea unui nod** constă în alocarea zonei de memorie în care se scrie nodul și legarea lui la structură (stabilirea legăturii cu predecesorul și cu succesorul lui).
- **Eliminarea unui nod** constă în ruperea nodului din structură, prin legarea predecesorului său cu succesorul lui, și eliberarea zonei de memorie pe care a ocupat-o.

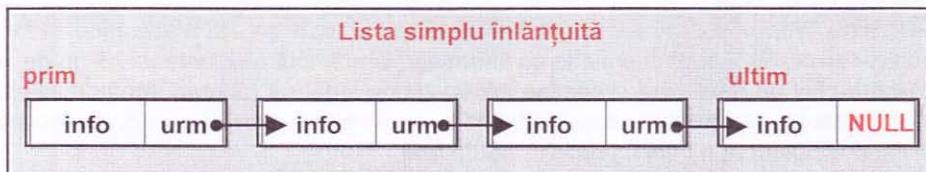
2.6.2. Clasificarea listelor



Algoritmii ce se pot folosi pentru prelucrarea listelor:

- inițializarea listei – se creează lista vidă;
- crearea listei – se adaugă repetat elemente la listă, pornind de la lista vidă;
- inserarea unui element în listă – la început, la sfârșit, în interior;
- ștergerea unui element din listă – la început, la sfârșit, în interior;
- parcurgerea listei – se vizitează elementele listei pentru a obține informații;
- căutarea în listă a unui element care îndeplinește anumite condiții;
- sortarea unei liste;
- concatenarea a două liste;
- divizarea în două liste.

2.6.3. Algoritmi pentru prelucrarea listelor simplu înăntărite



În implementarea algoritmilor următori se consideră că informația proprie-zisă este formată numai dintr-un câmp în care se memorează un număr întreg:

```

struct nod
    {int info;           //informația propriu-zisă
     nod *urm;};        //informația pentru legătură
nod *prim, *ultim, *p; //pointeri pentru exploatarea listei
int x;             //pentru valoarea ce se atribuie câmpului cu informații
    
```

În **lista vidă** atât nodul **prim** cât și nodul **ultim** nu există, și adresa lor are valoarea **NULL**: **prim = ultim = NULL;** Starea de **listă vidă** trebuie cunoscută atunci când se elimină noduri din listă, deoarece în lista vidă nu mai există noduri care să fie eliminate. Această stare se testează prin condiția: **prim == NULL**. Pentru testarea unei liste dacă este vidă se poate implementa funcția operand **este_vidă()** care va furniza valoarea 1 („adevărat”), dacă lista este vidă, și valoarea 0 („fals”) dacă lista nu este vidă.

```

int este_vidă(nod *prim)
    {return prim==NULL; }
    
```

2.6.3.1. Inițializarea listei

Prin acest algoritm se creează lista vidă. În acest caz, atât nodul **prim** cât și nodul **ultim** nu există, și li se atribuie adresa **NULL**.

Implementarea algoritmului. Se folosește funcția procedurală **init()** ai cărei parametri **prim** și **ultim** de tip **nod** se transmit prin referință, deoarece sunt parametri de ieșire.

```

void init(nod *&prim, nod *&ultim)
    {prim = ultim=NULL; }
    
```

2.6.3.2. Crearea listei

Deoarece în algoritmii de prelucrare trebuie să se cunoască adresa primului nod, este importantă **adăugarea primului nod la lista vidă**. Pașii algoritmului de creare a unei liste sunt:

PAS1. Se adaugă primul nod la listă (nodul **prim**).

PAS2. Cât timp mai există informație, execută: se adaugă un nod la listă.

2.6.3.3. Adăugarea primului nod la listă

În lista cu un singur nod, adresa de legătură a nodului **prim** are valoarea **NULL** – și atât nodul **prim** cât și nodul **ultim** au aceeași adresă. Pașii execuției în acest algoritm sunt:

PAS1. Se cere alocarea de memorie pentru nodul **prim**.

PAS2. Se scrie informația în nodul **prim**.

PAS3. Adresei de legătură a nodului **prim** i se atribuie valoarea **NULL**.

PAS4. Nodului **ultim** i se atribuie adresa nodului **prim**.

Implementarea algoritmului. Se folosește funcția procedurală **adauga_nod()** ai cărei parametri **prim** și **ultim** de tip **nod** se transmit prin referință, deoarece sunt parametri de ieșire.

```

void adaug_nod (nod *&prim, nod *&ultim)
    {prim = new nod; prim->info=x; prim->urm=NULL; ultim=prim; }
    
```

2.6.3.4. Adăugarea unui nod la listă

Pentru adăugarea unui nod p la listă, în funcție de cerințele problemei, se poate folosi unul dintre algoritmii următori:

1. adăugarea în fața primului nod;
2. adăugarea după ultimul nod;
3. adăugarea într-o poziție în interiorul listei.

Adăugare în fața primului nod

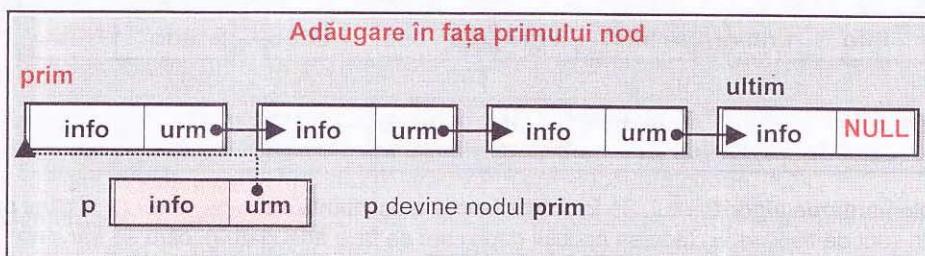
Pașii execuțiați în acest algoritm sunt:

PAS1. Se cere alocarea de memorie pentru nodul p .

PAS2. Se scrie informația în nodul p .

PAS3. Nodul p se leagă de nodul prim .

PAS4. Nodul p inserat devine nodul prim .



Implementarea algoritmului. Se folosește funcția procedurală `adauga_prim()` al cărei parametru prim de tip nod se transmite prin referință, deoarece este parametru de intrare-iesire.

```
void adauga_prim(nod *&prim)
{nod *p=new nod; p->info=x; p->urm=prim; prim=p;}
```

Adăugare după ultimul nod

Pașii execuțiați în acest algoritm sunt:

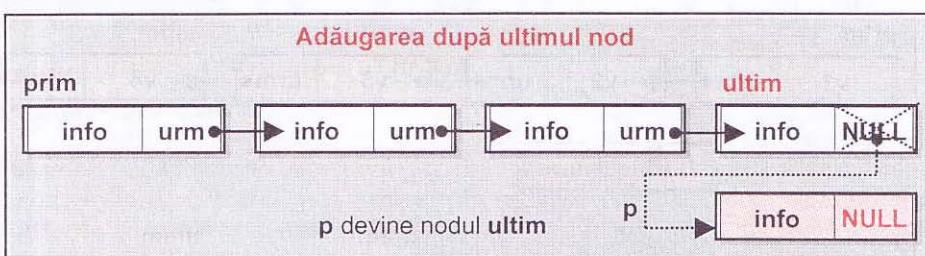
PAS1. Se cere alocarea de memorie pentru nodul p .

PAS2. Se scrie informația în nodul p .

PAS3. Nodul p este nod terminal (nu se leagă de nimic – adresa de legătură este NULL).

PAS4. Nodul ultim se leagă de nodul p adăugat.

PAS5. Nodul p adăugat devine nodul ultim .



Implementarea algoritmului. Se folosește funcția procedurală `adauga_ultim()` al cărei parametru ultim de tip nod se transmite prin referință, deoarece este parametru de intrare-iesire.

```
void adauga_ultim(nod *&ultim)
{nod *p=new nod; p->info=x; p->urm=NULL; ultim->urm=p; ultim=p;}
```

Adăugarea în interiorul listei se poate face în două moduri:

- după nodul cu adresa q;
- înainte de nodul cu adresa q.

Adăugarea în interiorul listei după nodul cu adresa q

Pașii algoritmului sunt:

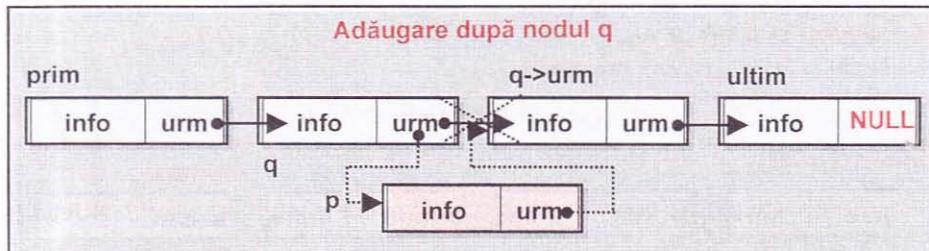
PAS1. Se cere alocarea de memorie pentru nodul p.

PAS2. Se scrie informația în nodul p.

PAS3. Nodul p se leagă de succesorul nodului q.

PAS4. Nodul q se leagă de nodul p adăugat.

PAS5. Dacă nodul q a fost ultimul nod, nodul p adăugat devine nodul **ultim**.

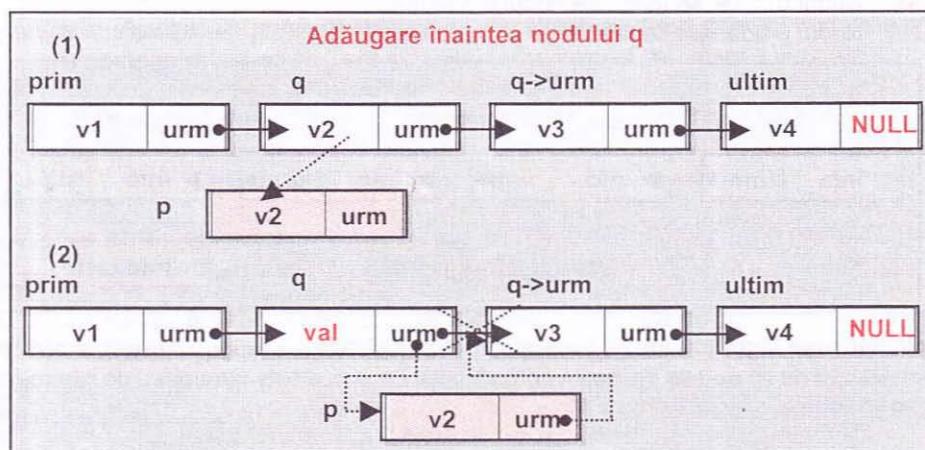


Implementarea algoritmului. Se folosește funcția procedurală `adauga_dupa()` ai cărei parametri sunt de tip `nod`: `q` (adresa nodului după care se face adăugarea), care se transmite prin valoare, deoarece este parametru de intrare, și `ultimo` (adresa ultimului nod), care se transmite prin referință, deoarece este parametru de intrare-iesire.

```
void adauga_dupa(nod *q, nod *&ultimo)
{nod *p=new nod;
 p->info=x; p->urm=q->urm; q->urm=p; if (q==ultimo) ultimo=p; }
```

Adăugarea în interiorul listei înaintea de nodul de adresă q

Pentru a adăuga nodul p înaintea nodului q, trebuie să legăm predecesorul nodului q de nodul p. Dar, predecesorul unui nod nu este cunoscut. Adăugarea unui nod în listă înseamnă, de fapt, inserarea în listă a informației pe care o conține, între alte două informații, și anume: informația din predecesorul nodului q trebuie să fie anterioară ei, iar informația din



nodul **q** trebuie să o urmeze. Astfel, în listă nu se va adăuga nodul **p** înainte de nodul **q**, ci după el, interschimbând apoi informațiile între cele două noduri. Pașii algoritmului sunt:

- PAS1.** Se cere alocarea de memorie pentru nodul **p**.
- PAS2.** Se copiază informația din nodul **q** în nodul **p**.
- PAS3.** Se scrie în nodul **q** informația care trebuie adăugată la listă.
- PAS4.** Nodul **p** se leagă de succesorul nodului **q**.
- PAS5.** Nodul **q** se leagă de nodul **p** adăugat.
- PAS6.** Dacă nodul **q** a fost ultimul nod, nodul **p** adăugat devine nodul **ultim**.

Implementarea algoritmului. Se folosește funcția procedurală **adauga_in_fata()** ai cărei parametri sunt de tip **nod**: **q** (adresa nodului înaintea căruia se face adăugarea), care se transmite prin valoare, deoarece este parametru de intrare, și **ultim** (adresa ultimului nod), care se transmite prin referință, deoarece este parametru de intrare-ieșire.

```
void adauga_in_fata(nod *q, nod *&ultim)
{nod *p=new nod; p->info=q->info; q->info=x; p->urm=q->urm; q->urm=p;
 if (q==ultim) ultim=p;}
```

2.6.3.5. Parcurgerea listei

Prin acest algoritm se vizitează fiecare nod al listei, pornind de la primul nod, până la ultimul nod, în ordinea de înlățuire a nodurilor – furnizată de adresa **urm** din nodul vizitat. Lista se parcurge pentru a prelucra informația stocată în noduri.

Implementarea algoritmului. Se folosește funcția procedurală **parcurge()** al cărei parametru **prim** de tip **nod** se transmite prin valoare, deoarece este parametru de intrare.

```
void parcuge(nod *prim)
{for (nod *p=prim; p!=NULL; p=p->urm)
 //se prelucreează p->info;}
```

Atât vectorul, cât și lista sunt structuri liniare, iar algoritmii de parcurs sunt asemănători:

	Vectorul	Lista
Variabila folosită pentru parcurs	i de tip int pentru indicele elementului curent din vector	p de tip pointer către tipul nod al listei pentru adresa elementului curent din listă
Inițializarea variabilei	i=0 – indicele primului element din vector	p=prim – adresa primului nod din listă
Trecerea la elementul următor	i=i+1 – se incrementează indicele	p=p->urm – pointerului i se atribuie adresa nodului următor
Condiția de terminare	i==n – indicele i are prima valoare mai mare decât cea a ultimului element	p==NULL – adresa memorată în pointerul p este constanta NULL

2.6.3.6. Căutarea unui nod în listă

Într-o listă se poate căuta:

- 1. Nodul care îndeplinește o anumită condiție**, pe care o notăm cu **conditie** și care este exprimată printr-o expresie logică ce conține cel puțin un câmp cu informație din nod; valoarea condiției este dependență de informația stocată în nod. **Algoritmul** este: se parcurge lista începând de la primul nod, în ordinea de înlățuire a nodurilor, până se găsește nodul care îndeplinește condiția sau până s-a ajuns la sfârșitul listei.
- 2. Nodul care se găsește într-o anumită poziție în listă** pe care o notăm cu **poz**. **Algoritmul** este: se parcurge lista începând de la primul nod, în ordinea de înlățuire a nodurilor, când s-au parcurs **poz** noduri sau până s-a ajuns la sfârșitul listei

Implementarea algoritmului. Se folosește o funcție operand cu tipul **pointer** către tipul **nod**. În ambele variante:

- parametrul funcției este **prim** de tip **nod** și se transmite prin valoare, deoarece este parametru de intrare;
- variabila locală **p** de tip pointer către **nod** se folosește pentru parcurgerea listei – este inițializată cu adresa primului nod;
- adresa nodului găsit se memorează în pointerul **p** care va fi returnată de funcție.

Varianta 1

```
nod *caut(nod *prim)
{for (nod *p=prim; p!=NULL && !conditie; p=p->urm); return p;}
```

Varianta 2 – Variabila locală **nr** de tip **int** se folosește pentru a număra pozițiile parcuse – este inițializată cu valoarea 1.

```
nod *caut(nod *prim, int poz)
{nod *p=prim; int nr=1;
 for (;p!=NULL && nr<poz; p=p->urm,nr++); return p;}
```

2.6.3.7. Eliminarea unui nod din listă

După eliminarea nodului din poziția **p** din listă se va elibera zona de memorie ocupată de nod. Eliminarea unui nod se face numai dacă lista nu este vidă. Pentru eliminarea unui nod din listă, în funcție de situație, se poate folosi unul dintre algoritmii următori:

1. eliminarea primului nod;
2. eliminarea ultimului nod;
3. eliminarea unui nod din interiorul listei.

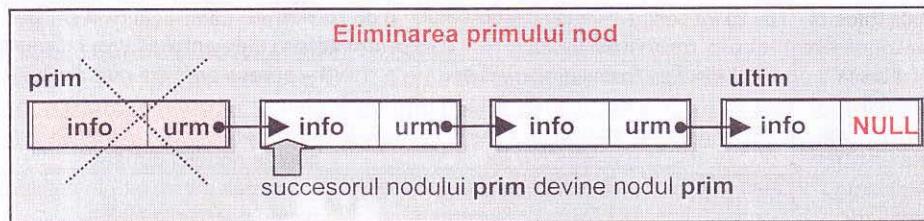
Eliminarea primului nod

Pașii execuției în acest algoritm sunt:

PAS1. Se salvează adresa nodului **prim** în pointerul **q**.

PAS2. Succesorul nodului **prim** devine nodul **prim**.

PAS3. Se cere eliberarea zonei de memorie de la adresa memorată în pointerul **q**.



Implementarea algoritmului. Se folosește funcția procedurală **elimina_prim()** al cărei parametru **prim** de tip **nod** se transmite prin referință, deoarece este parametru de intrare- ieșire.

```
void elimina_prim(nod *&prim)
{nod q=prim; prim=prim->urm; delete q;}
```

Eliminarea ultimului nod

Pașii execuției în acest algoritm sunt:

PAS1. Se salvează adresa nodului **ultim** în pointerul **q**.

PAS2. Se caută predecesorul ultimului nod, prin parcurgerea listei începând de la primul nod, până la predecesorul nodului terminal (nodul care nu se leagă de nimic – adresa de legătură are valoarea NULL).

PAS3. Predecesorul nodului **ultim** devine nod terminal (adresa de legătură este NULL).

PAS4. Predecesorul nodului **ultim** devine nodul **ultim**.

PAS5. Se cere eliberarea zonei de memorie de la adresa memorată în pointerul **q**.



Implementarea algoritmului. Se folosește funcția procedurală `elimina_ultim()` al cărei parametru **ultim** de tip nod se transmite prin referință, deoarece este parametru de intrare-iesire.

```
void elimina_ultim(nod *prim, nod *&ultim)
{nod *p, *q=ultim;
 for (p=prim; p->urm->urm!=NULL; p=p->urm);
 p->urm=NULL; ultim=p; delete q;}
```

Eliminarea unui nod din interiorul listei

Pentru a elimina nodul **p** aflat în interiorul listei, trebuie să legăm predecesorul nodului **p** de succesorul lui. Dar, predecesorul unui nod nu este cunoscut. Eliminarea unui nod din listă înseamnă de fapt eliminarea din listă a informației pe care o conține. Astfel, din listă nu se va elibera nodul **p**, ci succesorul său (care este cunoscut), după ce informația din el a fost copiată în nodul **p**. Pașii execuția în acest algoritm sunt:

- PAS1.** Se salvează adresa succesorului nodului **p** în pointerul **q**.
- PAS2.** Se copiază în nodul **p** toată informația din succesorul lui (informația propriu-zisă și informația de legătură).
- PAS3.** Se cere eliberarea zonei de memorie de la adresa memorată în pointerul **q**.
- PAS4.** Dacă succesorul nodului **p** era nodul **ultim**, atunci nodul **p** devine nodul **ultim**.

Implementarea algoritmului. Se folosește funcția procedurală `elimina()` ai cărei parametri sunt de tip **nod**: **p** (adresa nodului care se elimină), care se transmite prin valoare, deoarece este parametru de intrare, și **ultim**, care se transmite prin referință, deoarece este parametru de intrare-iesire.

```
void elimina(nod *p, nod *&ultim)
{nod *q=p->urm;
 p->info=p->urm->info; p->urm=p->urm->urm; delete q;
 if (p->urm==NULL) ultim=p;}
```

2.6.3.8. Eliberarea spațiului de memorie ocupat de listă

Dacă în cadrul unei aplicații care prelucrează mai multe liste, nu mai este necesară una dintre ele, se va elibera întregul spațiu ocupat de listă. Prin acest algoritm se vizitează fiecare nod al listei, pornind de la primul nod, până la ultimul nod, în ordinea de înlănțuire a nodurilor – furnizată de adresa **urm** din nodul vizitat, și se eliberează zona de memorie ocupată de fiecare nod. Pașii execuția în acest algoritm sunt:

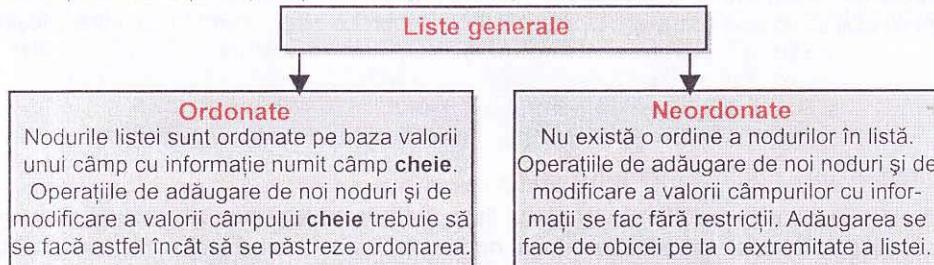
- PAS1.** Se initializează pointerul **p** cu adresa primului nod din listă – **prim** (**p** \leftarrow **prim**).
- PAS2.** Cât timp nu s-a parcurs toată lista (**p** \neq **NULL**) execută:
 - PAS3.** Se salvează adresa nodului **p** în pointerul **q**.
 - PAS4.** Se trece la succesorul nodului **p** (**p** \leftarrow **p** \rightarrow **urm**).
 - PAS5.** Se cere eliberarea zonei de memorie de la adresa memorată în pointerul **q**.
Se revine la Pasul 2.
- PAS6.** Primul nod nu mai are adresă alocată (**prim** \leftarrow **NULL**).

Implementarea algoritmului. Se folosește funcția procedurală `elibera()` al cărei parametru este `prim` de tip `nod`. Fiind parametru de intrare-iesire se transmite prin referință

```
void elibera(nod *&prim)
{ nod *p=prim, *q;
  while(p!=NULL) { q=p; p=p->urm; delete q; }
  prim=NULL; }
```

2.6.3.9. Liste ordonate

În funcție de cerințele aplicației listele generale pot fi clasificate astfel:



În cazul listelor ordonate, dacă informația din nodul listei este o dată elementară, cheia de ordonare va fi data elementară. Dacă informația din nodul listei este o înregistrare, cheia de ordonare va fi unul dintre câmpurile înregistrării. Pentru prelucrarea listelor ordonate puteți folosi:

- **algoritmul de sortare prin inserție**;
- **algoritmul de interclasare a două liste ordonate**.

Algoritmul de sortare prin inserție

Acest algoritm se folosește pentru întreținerea unei liste ordonate. Se pornește de la lista care conține un singur nod (considerată o listă ordonată) și orice adăugare a unui nod la listă se face într-o poziție prin care se păstrează ordonarea în listă. În implementarea algoritmului s-a folosit o listă pentru care câmpul cheie este de tip numeric întreg, iar ordonarea este crescătoare după valoarea câmpului cheie. Deoarece în acest algoritm nu este necesară informația despre adresa ultimului nod, din subprograme au fost eliminate prelucrările care se referă la acesta. Numerele care trebuie memorate în nodurile listei se citesc de la tastatură până se citește valoarea 0 (are semnificația că nu mai există informație de adăugat).

Varianta 1 – Se pornește de la lista care conține primul nod cu informație. Pașii execuției în acest algoritm sunt:

PAS1. Se adaugă primul nod cu informație la listă.

PAS2. Cât timp mai există informație de adăugat **execută**

PAS3. Se caută nodul înaintea căruia trebuie adăugat noul nod.

PAS4. Dacă s-a ajuns la sfârșitul listei, **atunci** se adaugă noul nod ca ultimul nod din listă; **altfel**, se adaugă noul nod în fața nodului găsit. Se revine la Pasul 2.

Implementarea algoritmului.

```
#include<iostream.h>
struct nod {int info;
            nod *urm;};
int n;
void adauga_nod(nod *&prim)
{prim=new nod; prim->info=n; prim->urm=NULL; }
```

```

void adauga_in_fata(nod *q)
{nod *p=new nod; p->urm=q->urm; p->info=q->info; q->info=n; q->urm=p;}
void adauga_ultim(nod *q)
{nod *p=new nod; p->info=n; q->urm=p; p->urm=NULL;}
nod * cauta(nod *prim)
{nod *q=prim;
 while(q->info<n && q->urm!=NULL) q=q->urm;
 return q;}
void afisare(nod *prim)
{for (nod *p=prim; p!=NULL; p=p->urm) cout<<p->info<<" ";}
void main()
{nod *prim,*q;
 cout<<"numar "; cin>>n;
 adauga_nod(prim); //Se adaugă primul nod
 cout<<"numar "; cin>>n;
 while(n!=0) //Cât timp mai există informație de adăugat
 {q=cauta(prim);
 //Se caută nodul q înaintea căruia trebuie adăugat nodul p
 if (q->info<n) //Dacă s-a ajuns la sfârșitul listei
     adauga_ultim(q); //nodul p se adaugă ca ultim nod
 else adauga_in_fata(q); //nodul p se adaugă în fața nodului q
 cout<<"numar "; cin>>n;}
 afisare(prim); //Se afișează informațiile din listă
}

```

Varianta 2 – Se pornește de la lista care conține un nod **santinelă** (un nod care marchează sfârșitul listei). Acest nod conține în câmpul cheie o informație care are o valoare care face ca el să fie întotdeauna ultimul nod din listă (de exemplu, în cazul unei liste ordonate crescător, în care câmpul folosit pentru ordonare este de tip **int**, se adaugă un nod care are în acest câmp cea mai mare valoare pentru tipul **int** – **MAXINT**). Când se va afișa informația din listă, ultimul nod din listă nu se mai afișează. Pașii execuției în acest algoritm sunt:

PAS1. Se adaugă nodul santinelă la listă.

PAS2. Cât timp există informație de adăugat **execută**

PAS3. Dacă informația care se adaugă trebuie să fie la începutul listei, **atunci** se adaugă noul nod ca primul nod din listă și se revine la Pasul 2; **altfel**, se trece la pasul următor.

PAS4. Se caută nodul înaintea căruia trebuie adăugat noul nod.

PAS5. Se adaugă noul nod în fața nodului găsit. Se revine la Pasul 2.

Implementarea algoritmului.

```

#include<iostream.h>
#include<values.h>
struct nod {int info;
            nod *urm;};
int n;
void adauga_santinela(nod *&prim)
{prim=new nod; prim->info=MAXINT; prim->urm=NULL;}
void adauga_prim(nod *&prim)
{nod *p=new nod; p->info=n; p->urm=prim; prim=p;}
void adauga_in_fata(nod *q)
{nod *p=new nod;
 p->urm=q->urm; p->info=q->info; q->info=n; q->urm=p;}

```

```

nod *cauta(nod *prim)
{ nod *p=prim;
  for (;p->info<n;p=p->urm); return p;}
void afisare(nod *prim) //Nu se afișează informația din ultimul nod
{for (nod *p=prim; p->urm!=NULL; p=p->urm) cout<<p->info<<" ";}
void main()
{ nod *prim,*q;
  adauga_santinela(prim); //Se adaugă nodul santinelă
  cout<<"numar "; cin>>n;
  while(n!=0) //Cât timp mai există informație de adăugat
    {if (n<prim->info) /*Dacă informația trebuie să fie la
                           începutul listei */
       adauga_prim(prim); //nodul p se adaugă ca prim nod
     else {q=cauta(prim);
           //Se caută nodul q înaintea căruia trebuie adăugat nodul p
           adauga_in_fata(q);} //nodul p se adaugă în fața nodului q
     cout<<"numar "; cin>>n;}
  afisare(prim);}

```

Algoritmul de interclasare a două liste ordonate

Acest algoritm folosește același principiu ca și algoritmul de interclasare a doi vectori ordonați. Cele două liste sunt ordonate după același câmp cheie și criteriu de ordonare și se obține o a treia listă care conține informația din primele două liste ordonată după același criteriu, folosind același câmp cheie. Pașii execuției în acest algoritm sunt:

- PAS1.** Se creează Lista 1 și Lista 2 ordonate după același criteriu și după același câmp cheie.
- PAS2.** Se adaugă primul nod la Lista 3 astfel: **dacă** în conformitate cu criteriul de ordonare folosit trebuie adăugată informația din primul nod al Listei 1, **atunci** se adaugă informația din nodul **prim** al Listei 1 și în Lista 1 se trece la succesorul nodului **prim**, iar în Lista 2 la nodul **prim**; **altfel**, se adaugă informația din nodul **prim** al Listei 2 și în Lista 2 se trece la succesorul nodului **prim**, iar în Lista 1 la nodul **prim**.
- PAS3.** Cât timp nu s-a ajuns la sfârșitul Listei 1 sau al Listei 2, **execută**:

 - PAS4.** Se adaugă la Lista 3 un nod după ultimul nod astfel: **dacă** în conformitate cu criteriul de ordonare folosit trebuie adăugată informația din nodul curent al Listei 1, **atunci** se adaugă informația din acest nod și în Lista 1 se trece la succesorul nodului curent; **altfel**, se adaugă informația din nodul curent al Listei 2 și în Lista 2 se trece la succesorul nodului curent. Se revine la Pasul 3.
 - PAS5.** **Dacă** s-a ajuns la sfârșitul Listei 1, **atunci** la Lista 3 se adaugă după ultimul nod nodurile rămase în Lista 2; **altfel**, la Lista 3 se adaugă după ultimul nod nodurile rămase în Lista 1.

Implementarea algoritmului. În implementarea algoritmului s-au folosit liste pentru care câmpul cheie este de tip numeric întreg, iar ordonarea este crescătoare după valoarea câmpului cheie. Numerele care trebuie memorate în nodurile listelor se citesc de la tastatură în variabila **n** până se citește valoarea 0 (are semnificația că nu mai există informație de adăugat). Din subprogramele în care nu este necesară informația despre adresa ultimului nod au fost eliminate prelucrările care se referă la acesta. Cele trei liste se identifică prin adresa primului nod (**prim1**, **prim2** și respectiv **prim3**). Deoarece în acest algoritm nu este necesară informația despre adresa ultimului nod, din subprograme au fost eliminate prelucrările care se referă la acesta. Pentru parcurgerea celor două liste se folosesc pointerii **q** și respectiv **r**, iar pentru lista care se creează prin interclasare – pointerul **p**.

```

#include<iostream.h>
struct nod {int info;
            nod *urm;};

int n;
void adauga_nod(nod *&prim)
{prim=new nod; prim->info=n; prim->urm=NULL; }
void adauga_in_fata(nod *q)
{nod *p=new nod;
 p->urm=q->urm; p->info=q->info; q->info=n; q->urm=p; }
void adauga_ultim(nod *q)
{nod *p=new nod; p->info=n; q->urm=p; p->urm=NULL; }
nod *cauta(nod *prim)
{nod *p=prim;
 while(p->info<n && p->urm!=NULL) p=p->urm;
 return p; }
void creare(nod *&prim) //Se creează o listă ordonată
{nod *q;
 cout<<"numar "; cin>>n; adauga_nod(prim);
 cout<<"numar "; cin>>n;
 while(n!=0)
 {q=cauta(prim);
 if (q->info<n) adauga_ultim(q); else adauga_in_fata(q);
 cout<<"numar "; cin>>n;}}
void afisare(nod *prim)
{nod *p=prim;
 while(p!=NULL) {cout<<p->info<<" "; p=p->urm;}}
void main()
{nod *prim1,*prim2,*prim3,*q,*r,*p;
 creare(prim1); creare(prim2);
 if (prim2->info>prim1->info)
     {n=prim1->info; q=prim1->urm; r=prim2; }
 else {n=prim2->info; r=prim2->urm; q=prim1; }
 adauga_nod(prim3); p=prim3;
 while(q!=0 && r!=0)
 {if (r->info>q->info) {n=q->info; q=q->urm; }
   else {n=r->info; r=r->urm; }
   adauga_ultim(p); p=p->urm; }
 if (q!=0)
     while (q!=0) {n=q->info; adauga_ultim(p); p=p->urm; q=q->urm; }
 else
     while (r!=0) {n=r->info; adauga_ultim(p); p=p->urm; r=r->urm; }
 afisare(prim3); } //Se afișează informațiile din lista interclasată

```

2.6.3.10. Prelucrarea listelor simplu înlățuită

Rezolvarea problemelor în care organizați datele sub formă de liste presupune folosirea algoritmilor prezentați, astfel:

1. Se creează lista prin folosirea următorilor algoritmi:

- Se adaugă primul nod la lista vidă (**algoritmul pentru adăugarea la listă a primului nod**).
- Se adaugă câte un nod la listă. Poziția în care se adaugă nodul depinde de cerințele problemei. Dacă lista nu este ordonată, adăugarea se poate face la sfârșitul

sau la începutul listei (**algoritmul pentru adăugare la începutul sau la sfârșitul listei**). Dacă lista este ordonată, se parcurge mai întâi lista pentru a găsi poziția de inserare (**algoritmul pentru căutarea unui nod în listă**) după care se inserează noul nod în poziția găsită (**algoritmul pentru adăugare în interiorul listei**).

2. Se întreține lista prin folosirea următorilor algoritmii:

- Se adaugă noi noduri la listă. Ca și la crearea listei, în funcție de cerințele problemei se va folosi unul dintre **algoritmii de adăugare**.
- Se elimină noduri din listă. Mai întâi se parcurge lista pentru a găsi nodul care se elimină (**algoritmul pentru căutarea unui nod în listă**) după care se elimină nodul din poziția găsită folosind unul dintre **algoritmii pentru eliminare** – în funcție de poziția în care a fost găsit nodul.
- Se modifică informația dintr-un nod al listei. Mai întâi se parcurge lista pentru a găsi nodul în care se va modifica informația (**algoritmul pentru căutarea unui nod în listă**). Dacă lista nu este ordonată sau dacă informația care se modifică nu face parte dintr-un câmp cheie dintr-o listă ordonată, se modifică valoarea câmpului respectiv. Dacă lista este ordonată și, prin modificarea valorii câmpului, se poate distruge această ordonare, se modifică valoarea câmpului, se salvează informația din nodul listei într-o variabilă intermediară, se elimină din vechea poziție (**algoritmul pentru eliminarea unui nod din listă**), se parcurge lista pentru a găsi noua poziție (**algoritmul pentru căutarea unui nod în listă**) și se inserează nodul în poziția găsită (**algoritmul pentru adăugarea unui nod la listă**).

3. Se obțin informații din listă.

Se parcurge lista (**algoritmul de parcurs a listei**) și se vizitează fiecare nod al listei pentru a extrage din el informațiile necesare.

Studiu de caz

Scop: exemplificarea modului în care, pentru rezolvarea problemei, folosiți algoritmii de prelucrare a listelor simplu înlănțuite și implementarea lor cu ajutorul subprogramelor.

Enunțul problemei 1. Se citește dintr-un fișier text un sir de numere separate prin spațiu cu care se creează o listă simplu înlănțuită în ordinea în care sunt citite numerele din fișier. Să se verifice dacă lista conține numai numere distincte și apoi să se adauge după fiecare număr impar din listă valoarea 100 și să se afișeze numerele din listă. Pentru testarea programului se vor folosi două seturi de numere: {2, 5, 10, 3, 8} și {2, 5, 10, 5, 8, 7}.

Nodurile listei nu sunt ordonate conform unui criteriu. Numerele se vor citi din fișier și se vor scrie în listă prin adăugare după ultimul nod. Problema se descompune în următoarele subprobleme, iar algoritmul de rezolvare a unei subprobleme este implementat cu ajutorul unui subprogram:

- [P1] Se citește primul număr din fișier și se adaugă primul nod la listă (nodul **prim**) – subprogramul **adauga_nod()**.
- [P2] Cât timp mai există numere în fișier **execută**: se citește un număr din fișier și se adaugă un nod cu numărul respectiv după ultimul nod din listă – subprogramul **adauga_ultim()**.
- [P3] Se verifică dacă numerele din listă sunt distincte, astfel: se parcurge lista de la primul nod până la penultimul nod și se verifică dacă numărul din nodul curent mai există în nodurile care urmează după el până la sfârșitul listei – subprogramul **distincte()**.
- [P4] Se parcurge lista de la primul nod până la sfârșitul ei și dacă numărul din nod este impar se adaugă după el un nod care conține valoarea 100 – subprogramul **prelucrare()**.

- P5** Se parurge lista de la primul nod până la sfârșitul ei și se afișează informația din fiecare nod – subprogramul **afisare()**.

```
#include<fstream.h>
struct nod {int info;
            nod *urm;};
fstream f("listal.txt",ios::in);
int n;
void adauga_nod(nod *&prim, nod *&ultim)
{prim=new nod; prim->info=n; prim->urm=NULL; ultim=prim;}
void adauga_ultim(nod *&ultim)
{nod *p=new nod; p->info=n; p->urm=NULL; ultim->urm=p; ultim=p;}
void adauga_dupa(nod *&q, nod *&ultim)
{nod *p=new nod; p->info=100; p->urm=q->urm; q->urm=p;
 if (q==ultim) ultim=p;}
int distincte(nod *prim)
{for (nod *p=prim; p->urm!=NULL; p=p->urm)
    for (nod *q=p->urm; q!=NULL; q=q->urm)
        if (p->info==q->info) return 0;
return 1;}
void prelucrare(nod *prim, nod *&ultim)
{for (nod *p =prim; p!=NULL; p=p->urm)
    if (p->info%2==1) adauga_dupa (p,ultim);}
void afisare(nod *prim)
{for (nod *p=prim; p!=NULL; p=p->urm) cout<<p->info<<" ";
 cout<<endl;}
void main()
{nod *prim, *ultim; f>>n adauga_nod(prim,ultim);
 while (f>>n) adauga_ultim(ultim); f.close();
 if (distincte(prim)) cout<<"Elementele sunt distințe"<<endl;
 else cout<<"Elementele nu sunt distințe"<<endl;
 prelucrare(prim,ultim); afisare(prim);}
```

Enunțul problemei 2. Se citește dintr-un fișier text un număr cu maxim 20 de cifre. Să se verifice dacă numărul este palindrom. Pentru testarea programului se vor folosi două numere: 1234321 și 12345.

Se vor crea două liste: una cu cifrele numărului citite din fișier, prin adăugare după ultimul nod, iar a doua, cu cifrele memorate în nodurile primei liste prin adăugare în fața primului nod și se vor compara cele două liste. Din fișier se va citi fiecare cifră a numărului într-o variabilă de tip **char** și se va obține cifra scăzând din codul ASCII al caracterului cunoscut codul ASCII al cifrei 0. Cele două liste se identifică prin adresa primului nod (**prim1** și respectiv **prim2**) și prin adresa ultimului nod (**ultim1** și respectiv **ultim2**). Problema se descompune în următoarele subprobleme, iar algoritmul de rezolvare a unei subprobleme este implementat cu ajutorul unui subprogram

- P1** Se citește primul număr din fișier și se adaugă primul nod la Lista 1 (nodul **prim1**) – subprogramul **adauga_nod()**.

- P2** Cât timp mai există caractere în fișier, execută: se citește un caracter din fișier, se convertește în cifră și se adaugă un nod cu cifra respectivă după ultimul nod din Lista 1 – subprogramul **adauga_ultim()**.

- P3** Se adaugă primul nod la Lista 2 (nodul **prim2**) care va conține informația din primul nod al Listei 1 (**prim1**) – subprogramul **adauga_nod()**.

- P4** Se parcurge Lista 1 de la successorul primului nod până la sfârșitul ei și se adaugă un nod cu cifra respectivă înaintea primului nod din Lista 2 – subprogramul `adauga_prim()`.
- P5** Se parcurează simultan ambele liste de la primul nod până la sfârșit și se verifică dacă sunt egale cifrele memorate în nodul curent din cele două liste – subprogramul `palindrom()`.

```
#include<iostream.h>
struct nod {int info;
            nod *urm;};
fstream f("lista2.txt",ios::in);
int n;
void adauga_nod(nod *&prim, nod *&ultim)
{prim=new nod; prim->info=n; prim->urm=NULL; ultim=prim;}
void adauga_ultim(nod *&ultim)
{nod *p=new nod; p->info=n; p->urm=NULL; ultim->urm=p; ultim=p;}
void adauga_prim(nod *&prim)
{nod *p=new nod; p->info=n; p->urm=prim; prim=p;}
int palindrom(nod *prim1, nod *prim2)
{nod *p,*q;
 for (p=prim1,q=prim2; p->urm!=NULL; p=p->urm,q=q->urm)
     if (p->info!=q->info) return 0;
 return 1;}
void main()
{char c; nod *prim1,*ultim1,*prim2,*ultim2,*p;
 f>>c; n=c-'0'; adauga_nod(prim1,ultim1);
 while (f>>c) {n=c-'0'; adauga_ultim(ultim1);} f.close();
 n=prim1->info; adauga_nod(prim2,ultim2);
 for (p=prim1->urm; p!=NULL; p=p->urm)
 {n=p->info; adauga_prim(prim2);}
 if (palindrom(prim1,prim2)) cout<<"Este palindrom";
 else cout<<"Nu este palindrom";}
```

Enunțul problemei 3. Se citește dintr-un fișier text un sir de numere separate prin spațiu cu care se creează o listă simplu înlănțuită în ordinea în care sunt citite numerele din fișier. Se mai citește de la tastatură un număr **x**. Să se caute și să se steargă din listă nodul care conține acest număr. Pentru testarea programului se va folosi sirul de numere: {2, 5, 10, 3, 8}, iar pentru **x** patru valori: 2, 5, 8 și 7.

Nodurile listei nu sunt ordonate conform unui criteriu. Numerele se vor citi din fișier și se vor scrie în listă prin adăugare după ultimul nod. Problema se descompune în următoarele subprobleme, iar algoritmii pentru rezolvarea subproblemelor sunt implementați cu ajutorul subprogramelor:

- P1** Se citește valoarea pentru **x** de la tastatură.
- P2** Se citește primul număr din fișier și se adaugă primul nod la listă (nodul **prim**) – subprogramul `adauga_nod()`.
- P3** Cât timp mai există numere în fișier **execută**: se citește un număr din fișier și se adaugă un nod cu numărul respectiv după ultimul nod din listă – subprogramul `adauga_ultim()`.
- P4** Dacă primul nod din listă conține numărul **x**, atunci se elimină primul nod (subprogramul `elimina_prim()`); altfel, se caută predecesorul nodului care conține numărul **x** (subprogramul `cauta()`) și dacă se găsește acest nod, atunci se sterge nodul care urmează după el (subprogramul `elimina_urm()`).

- P5** Se parcurge lista de la primul nod până la sfârșitul ei și se afișează informația din fiecare nod – subprogramul **afisare()**.

```
#include<iostream.h>
struct nod {int info;
            nod *urm;};
fstream f("lista3.txt",ios::in);
int n,x;
void adauga_nod(nod *&prim, nod *&ultim)
{prim=new nod; ultim=prim; prim->info=n; prim->urm=NULL; }
void adauga_ultim(nod *&ultim)
{nod *p=new nod; p->info=n; p->urm=NULL; ultim->urm=p; ultim=p;}
nod * cauta(nod *p)
{while(p->urm!=NULL && p->urm->info!=x) p=p->urm;
 return p;}
void elibera_prim(nod *&prim)
{nod *q=prim; prim=prim->urm; delete q;}
void elibera_urm(nod *p)
{nod *q=p->urm; p->urm=q->urm; delete q;}
void afisare(nod *prim)
{for (nod *p=prim;p!=NULL;p=p->urm) cout<<p->info<<" ";
 cout<<endl;}
void main()
{nod *prim,*ultim,*p; cout<<"x= "; cin>>x;
 f>>n; adauga_nod(prim,ultim);
 while(f>>n) adauga_ultim(ultim); f.close();
 if (prim->info==x) elibera_prim(prim);
 else {p=cauta(prim);
        if (p->urm!=NULL) elibera_urm(p);}
 afisare(prim);}
```

Enunțul problemei 4. Se citește dintr-un fișier text un sir de numere separate prin spațiu cu care se creează o listă simplu înlănțuită în ordinea în care sunt citite numerele din fișier. Să se eliminate din listă valorile care se repetă, în afară de prima lor apariție. Pentru testarea programului se va folosi sirul de numere: {2, 4, 9, 9, 9, 6, 2, 9, 9}.

Nodurile listei nu sunt ordonate conform unui criteriu. Numerele se vor citi din fișier și se vor scrie în listă prin adăugare după ultimul nod. Problema se descompune în următoarele subprobleme, iar algoritmii pentru rezolvarea subproblemelor sunt implementați cu ajutorul subprogramelor:

- P1** Se citește primul număr din fișier și se adaugă primul nod la listă (nodul **prim**) – subprogramul **adauga_nod()**.

- P2** Cât timp mai există numere în fișier **execută**: se citește un număr din fișier și se adaugă un nod cu numărul respectiv după ultimul nod din listă – subprogramul **adauga_ultim()**.

- P3** Se elimină din listă valorile care se repetă, astfel (subprogramul **prelucrare()**):
Pentru un pointer **p** care indică fiecare nod din listă, începând cu primul nod până la penultimul nod, **execută**:

- Se inițializează pointerul **q** cu adresa nodul curent (indicat de pointerul **p**).
- Cât timp pointerul **q** nu indică ultimul nod, **execută**: dacă nodul indicat de pointerul **p** conține același număr cu succesorul nodului indicat de pointerul **q**,

atunci se elimină succesorul nodului `q` (- subprogramul `elimina_urm()`); altfel, pointerul `q` indică succesorul nodului.

- P4** Se parcurge lista de la primul nod până la sfârșitul ei și se afișează informația din fiecare nod (subprogramul `afisare()`).

```
#include<iostream.h>
struct nod {int info;
            nod *urm;};
fstream f("lista4.txt",ios::in);
int n;
void adauga_nod(nod *&prim, nod *&ultim)
{prim=new nod; prim->info=x; prim->urm=NULL; ultim=prim;}
void adauga_ultim(nod *&ultim)
{nod *p=new nod; p->info=x; p->urm=NULL; ultim->urm=p; ultim=p;}
void elimina_urm(nod *p)
{nod *q=p->urm; p->urm=q->urm->urm; delete q;}
void prelucrare(nod *prim)
{nod *p,*q;
for (p=prim;p->urm!=NULL;p=p->urm)
{n=p->info; q=p;
while (q!=NULL)
{if (q->urm!=NULL && q->urm->info==n) elimina_urm(q);
else q=q->urm;}}}
void afisare(nod *prim)
{for (nod *p=prim; p!=NULL;p=p->urm) cout<<p->info<<" ";
cout<<endl;}
void main() {nod *prim,*ultim; f>>n; adauga_nod(prim,ultim);
while (f>>n) adauga_ultim(ultim); f.close();
prelucrare(prim); afisare(prim);}
```

Enunțul problemei 5. Adunarea a două polinoame. Se citesc dintr-un fișier text de pe prima linie un număr **n1**, care reprezintă numărul de coeficienți nenuli ai unui polinom, apoi de pe următoarele **n1** linii coeficienții nenuli și gradul, de pe linia următoare un număr **n2** care reprezintă numărul de coeficienți nenuli ai celui de al doilea polinom și apoi de pe următoarele linii coeficienții nenuli și gradul. Să se creeze cu aceste informații două liste simplu înlántuite și să se adune cele două polinoame. Coeficientul și gradul fiecărui termen din polinomul sumă se vor salva într-un fișier. Pentru testarea programului se vor folosi polinoamele $-10x^4+5x^2-3x$ și $3x^3+7x^2+x+2$.

Se vor crea două liste în care se vor memora cele două polinoame. Informația utilă va fi memorată într-o înregistrare cu două câmpuri: un câmp pentru coeficient și un câmp pentru grad. Ambele liste se creează prin citirea datelor din fișier și adăugare după ultimul nod. În a treia listă se va memora polinomul obținut prin adunarea celor două polinoame. Cele trei liste se identifică prin adresa primului nod (`prim1`, `prim2` și respectiv `prim3`) și prin adresa ultimului nod (`ultim1`, `ultim2` și respectiv `ultim3`). Problema de creare a listei sumă a polinoamelor se descompune în următoarele subprobleme, iar algoritmii pentru rezolvarea subproblemelor sunt implementații cu ajutorul subprogramelor:

- P1** Se citesc din fișier **n1** și coeficientul și gradul primului termen din primul polinom. Se adaugă primul nod la lista L1 (nodul `prim1`) – subprogramul `adauga_nod()`.
- P2** Pentru următoarele **n1-1** perechi de numere din fișier, execută: se citesc din fișier coeficientul și gradul unui termen al primului polinom și se adaugă un nod cu informația respectivă după ultimul nod din lista L1 – subprogramul `adauga_ultim()`.

- P3** Se citesc din fișier **n2** și coeficientul și gradul primului termen din al doilea polinom.
Se adaugă primul nod la lista L2 (nodul **prim2**) – subprogramul **adauga_nod()**.
- P4** Pentru următoarele **n2-1** perechi de numere din fișier, **execută**: se citesc din fișier coeficientul și gradul unui termen al celui de al doilea polinom și se adaugă un nod cu informația respectivă după ultimul nod din lista L2 – subprogramul **adauga_ultim()**.
- P5** Se adaugă primul nod (**prim3**) la lista L3 cu următoarea informație: **dacă** gradul din primul nod din lista L1 este egal cu gradul primului nod din lista L2, **atunci** gradul este egal cu gradul nodului din Lista 1, iar coeficientul este egal cu suma coeficienților din nodurile celor două liste și în ambele liste se trece la nodul următor; **altfel**, **dacă** gradul din primul nod din lista L1 este mai mare decât gradul primului nod din lista L2, **atunci** gradul este egal cu gradul nodului din Lista 1, iar coeficientul este egal cu coeficientul nodului din Lista 1 și în Lista 1 se trece la următorul nod; **altfel**, gradul este egal cu gradul nodului din Lista 2, iar coeficientul este egal cu coeficientul nodului din Lista 2 și în Lista 2 se trece la următorul nod – subprogramul **adauga_nod()**.
- P6** Cât timp nu s-a ajuns la sfârșitul Listei 1 și al Listei 2, **execută**: se adaugă un nod la lista L3 după ultimul nod, cu următoarea informație: **dacă** gradul din nodul curent din lista L1 este egal cu gradul nodului curent din lista L2, **atunci** gradul este egal cu gradul nodului din Lista 1, iar coeficientul este egal cu suma coeficienților din nodurile celor două liste și în ambele liste se trece la nodul următor; **altfel**, **dacă** gradul din nodul curent din lista L1 este mai mare decât gradul din nodul curent din lista L2, **atunci** gradul este egal cu gradul nodului din Lista 1, iar coeficientul este egal cu coeficientul nodului din Lista 1 și în Lista 1 se trece la următorul nod; **altfel**, gradul este egal cu gradul nodului din Lista 2, iar coeficientul este egal cu coeficientul nodului din Lista 2 și în Lista 2 se trece la următorul nod – subprogramul **adauga_ultim()**.

```
#include<iostream.h>
fstream fl("lista5.txt",ios::in),f2("polinom.txt",ios::out);
struct nod {int c,g;
            nod *urm;};
int c,g,n;
void adauga_nod(nod *&prim,nod *&ultim)
{prim=new nod; ultim=prim; prim->urm=NULL; prim->c=c; prim->g=g;}
void adauga_ultim(nod *&ultim)
{nod *p=new nod; p->urm=NULL; ultim->urm=p; ultim=p; p->c=c; p->g=g;}
void creare(nod *&prim,nod *&ultim)
{fl>>c>>g; adauga_nod(prim,ultim);
 for (int i=2;i<=n;i++) {fl>>c>>g; adauga_ultim(ultim);}
void adunare(nod *prim1, nod *prim2, nod *&prim3, nod *&ultim3)
{nod *p=prim1,*q=prim2;
 if (p->g==q->g) {q=p->g; c=p->c+q->c; p=p->urm; q=q->urm;}
 else if (p->g>q->g) {g=p->g; c=p->c; p=p->urm;}
 else {g=q->g; c=q->c; q=q->urm;}
 adauga_nod(prim3,ultim3);
while(q!=NULL && p!=NULL)
{if (p->g==q->g) {g=p->g; c=p->c+q->c; p=p->urm; q=q->urm;}
 else if (p->g>q->g) {g=p->g; c=p->c; p=p->urm;}
 else {g=q->g; c=q->c; q=q->urm;}
 adauga_ultim(ultim3);}
if (p!=NULL)
 while(p!=NULL) {g=p->g; c=p->c; adauga_ultim(ultim3);}
else
```

```

while(q!=NULL) {q=q->g; c=q->c;adauga_ultim(ultim3);}
void afisare(nod *prim)
{for (nod *p=prim;p!=NULL;p=p->urm) cout<<p->c<<" "<<p->g<<endl;
cout<<endl;}
void salvare(nod *prim)
{for (nod *p=prim;p!=NULL;p=p->urm) f2<<p->c<<" "<<p->g<<endl;}
void main()
{nod *prim1,*ultim1,*prim2,*ultim2,*prim3,*ultim3;
int n1,n2;
f1>>n1; n=n1; creare(prim1,ultim1); afisare(prim1);
f1>>n2; n=n2; creare(prim2,ultim2); afisare(prim2);
adunare(prim1,prim2,prim3,ultim3); afisare(prim3); salvare(prim3);
f1.close(); f2.close();}

```

Enunțul problemei 6. Reuniunea și intersecția a două mulțimi. Se citesc dintr-un fișier text de pe prima linie un număr **n1**, care reprezintă numărul de elemente ale primei mulțimi, apoi de pe următoarea linie elementele mulțimii, de pe linia următoare un număr **n2** – numărul de elemente ale celei de a doua mulțimi, apoi de pe următoarea linie elementele mulțimii. Să se determine reuniunea și intersecția celor două mulțimi. Pentru testarea programului se vor folosi două seturi de date de intrare: mulțimile $A=\{1,2,3,4,5\}$ și $B=\{4,5,6,7\}$ și mulțimile $A=\{1,2,3\}$ și $B=\{4,5\}$.

Se vor crea două liste în care se vor memora cele două mulțimi. Ambele liste se creează prin citirea datelor din fișier și adăugare după ultimul nod. În a treia listă se va memora reuniunea celor două mulțimi, iar în a patra listă – intersecția. Cele patru liste se identifică prin adresa primului nod (**prim1**, **prim2**, **prim3** și respectiv **prim4**) și prin adresa ultimului nod (**ultim1**, **ultim2**, **ultim3** și respectiv **ultim4**).

Pentru determinarea reuniunii se vor executa următorii pași:

- Pas1. Se adaugă primul nod la Lista 3 (nodul **prim3**) care conține numărul din primul nod al Listei 1 (**prim1**).
- Pas2. Se parcurge Lista 1 de la successorul primului nod până la sfârșitul ei și se adaugă un nod cu numărul respectiv după ultimul nod din Lista 3.
- Pas3. Pentru fiecare nod din Lista 2, execută: se parcurge Lista 1 și dacă numărul din nodul curent din Lista 2 nu se găsește în Lista 1, atunci se adaugă un nod cu numărul respectiv după ultimul nod din Lista 3.

Pentru determinarea intersecției se vor executa următorii pași:

- Pas1. Se initializează Lista 4 ca listă vidă (**prim4=NULL** și **ultim4=NULL**).
- Pas2. Pentru fiecare nod din Lista 1, execută: se parcurge Lista 2 și dacă numărul din nodul curent din Lista 2 se găsește în Lista 1, atunci se adaugă un nod cu numărul respectiv ca prim nod în Lista 4 (nodul **prim4**).
- Pas3. Dacă s-a adăugat primul nod la Lista 4, atunci pentru un nod din Lista 1 de la successorul nodului curent până la ultimul nod, execută: se parcurge Lista 2 și dacă numărul din nodul curent din Lista 1 se găsește în Lista 2, atunci se adaugă în Lista 4, un nod cu numărul respectiv, după ultimul nod.

```

#include<iostream.h>
struct nod {int info;
            nod *urm;};
fstream f("lista6.txt",ios::in);
int x;
void adauga_nod(nod *&prim,nod *&ultim)

```

```

{prim=new nod; prim->info=x; prim->urm=NULL; ultim=prim;}
void adauga_ultim(nod *&ultim)
{nod *p=new nod; p->info=x; p->urm=NULL; ultim->urm=p; ultim=p;}
void creare(nod *&prim, nod *&ultim)
{int n; f>>n>>x; adauga_nod(prim,ultim);
 for (int i=2;i<=n;i++) {f>>x; adauga_ultim(ultim);}}
void reuniune(nod *prim1, nod *prim2, nod *&prim3, nod *&ultim3)
{nod *p,*q; int gasit;
 x=prim1->info; adauga_nod(prim3,ultim3);
 for(p=prim1->urm;p!=NULL;p=p->urm) {x=p->info; adauga_ultim(ultim3);}
 for (p=prim2;p!=NULL;p=p->urm)
 {for(q=prim1,gasit=0; q!=NULL && !gasit; q=q->urm)
 if(p->info==q->info) gasit=1;
 if(!gasit) {x=p->info; adauga_ultim(ultim3);}}
void intersectie(nod *prim1, nod *prim2, nod *&prim4, nod *&ultim4)
{nod *p,*q; int gasit; prim4=NULL; ultim4=NULL;
 for(gasit=0,p=prim1;p!=NULL && !gasit; p=p->urm)
 for (q=prim2; q!=NULL && !gasit; q=q->urm)
 if (p->info==q->info) {gasit=1; x=p->info;}
 if (gasit)
 {adauga_nod(prim4,ultim4);
 for(;p!=0;p=p->urm)
 {for (gasit=0,q=prim2; q!=NULL && !gasit; q=q->urm)
 if(p->info==q->info) gasit=1;
 if(gasit) {x=p->info; adauga_ultim(ultim4);}}}
void afisare(nod *prim)
{for (nod *p=prim; p!=NULL ;p=p->urm) cout<<p->info<<" ";
 cout<<endl;}
void main()
{nod *prim1,*ultim1,*prim2,*ultim2,*prim3,*ultim3,*prim4,*ultim4;
 creare(prim1,ultim1); creare(prim2,ultim2); f.close();
 reuniune(prim1,prim2,prim3,ultim3); cout<<"Reuniunea= "; afisare(prim3);
 intersectie(prim1,prim2,prim4,ultim4); cout<<"Intersectia= ";
 if (prim4!=NULL) afisare(prim4);
 else cout<<"Multimea vida";}

```


Temă


Scripti căte un program care să rezolve cerințele fiecărei probleme. Fiecare problemă se va descompune în subprobleme și algoritmul pentru rezolvarea unei subprobleme se va implementa cu un subprogram. Datele se transmit între subprograme cu ajutorul parametrilor de comunicație și nu al variabilelor globale. După executarea unei operații de prelucrare a listei simplu înălțătuite, se vor afișa numerele din nodurile listei pentru a se verifica dacă operația de prelucrare s-a executat corect. Se vor alege seturi de date de intrare astfel încât să se verifice algoritmul pe toate traseele lui. Pentru următorii 18 itemi în nodurile listelor se memorează numere întregi. Sirul de numere se citește dintr-un fișier text în care sunt scrise pe același rând, separate prin spațiu.

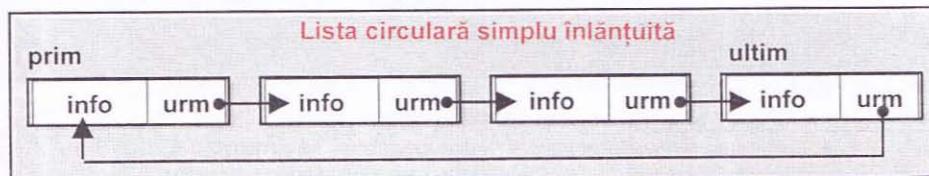
1. Se creează o listă în care ordinea de acces este cea în care sunt citite numerele. Se mai citesc de la tastatură două numere x și y . Se inserează în listă numărul y înaintea numărului x .
2. Se creează o listă în care ordinea de acces este inversă celei în care sunt citite numerele. Se elimină din listă cel mai mic număr și cel mai mare număr.

3. Se creează o listă și se afișează în ordinea inversă citirii din fișier numai numerele pare.
4. Se creează o listă în care ordinea de acces este cea în care sunt citite numerele din fișier. Se mai citește un număr n de la tastatură. Se afișează elementul cu numărul de ordine n din listă. Dacă nu există, se afișează un mesaj de informare.
5. Se creează o listă în care ordinea de acces este cea în care sunt citite numerele din fișier. Se inserează înaintea fiecărui număr factorii săi primi.
6. Se creează o listă în care ordinea de acces este cea în care sunt citite numerele din fișier. Se inserează între fiecare pereche de noduri cel mai mare divizor comun al celor două numere.
7. Se creează două liste în care ordinea de acces este cea în care sunt citite numerele din fișiere. Se creează a treia listă prin concatenarea listei care are cele mai puține elemente la lista care are cele mai multe elemente. Dacă listele au același număr de elemente, se va adăuga lista a doua la prima listă.
8. Se creează două liste în care ordinea de acces este cea în care sunt citite numerele din fișiere. Se creează a treia listă prin concatenarea celei de a doua liste la prima listă. Se elimină din a treia listă numerele care se repetă.
9. Se creează o listă în care ordinea de acces este cea în care sunt citite numerele din fișier. Se divizează lista în două liste: una care conține numere care sunt palindrom și una care conține numerele care nu sunt palindrom. Se salvează lista cu numere palindrom într-un alt fișier. Dacă nu au existat numere palindrom, în fișier se va scrie un mesaj de informare. În lista care nu conține numere palindrom se inserează după fiecare număr inversul său.
10. Se creează o listă în care ordinea de acces este cea în care sunt citite numerele din fișier. Se afișează numerele care au ultimele trei cifre identice, se elimină din listă numerele care au ultimele trei cifre consecutive și se inserează valoarea 10 înaintea numerelor care au suma ultimelor trei cifre egală cu 10.
11. Se creează o listă în care ordinea de acces este cea în care sunt citite numerele din fișier. Se afișează numerele care au mai mult de doi divizori primi, se inserează divizorii proprii în fața numerelor care au mai mult de trei divizori proprii și se elimină din listă numerele care au cel puțin două cifre identice.
12. Se creează o listă numai cu numerele prime din fișier. Se afișează cel mai mare număr prim și cel mai mic număr prim. Se verifică dacă lista conține numai numere distincte și se afișează un mesaj de informare. Dacă lista nu conține numai numere distincte, se elimină numere din listă astfel încât să conțină numai numere distincte. Se salvează lista creată într-un alt fișier. Dacă nu au existat numere prime în fișier, se va scrie un mesaj de informare. (**Indicație.** Se va crea o listă ordonată crescător și se vor afișa numerele prime din primul nod și din ultimul nod.)
13. Se creează două liste cu numerele citite din fișiere. În primul fișier numerele sunt ordonate crescător, iar în al doilea fișier numerele sunt ordonate descrescător. Se creează a treia listă prin interclasarea primelor două.
14. Se creează o listă în care ordinea de acces este cea în care sunt citite numerele din fișier. Să se inverseze ordinea de acces în listă, astfel încât parcurgerea să se facă de la ultimul număr către primul număr. (**Indicație.** Se mută ultimul nod la începutul listei și apoi, până se ajunge la numărul memorat la adresa care a fost a primului nod se inserează nodul **ultim** după ultimul nod inserat.)
15. Se creează o listă în care ordinea de acces este cea în care sunt citite numerele din fișier. Să se afișeze în ordine inversă numerele din listă. (**Indicație.** Se implementează un algoritm recursiv de parcurgere a listei.)

16. Se creează o listă ordonată cu numerele din fișier și se divizează apoi lista în două liste: una cu numere pare și una cu numere impare.
17. În fișier sunt memorate foarte multe numere (maxim 10.000). Foarte multe dintre aceste numere se repetă (există maxim 100 de numere distincte). Se creează o listă ordonată crescător numai cu numerele distincte și cu frecvența lor de apariție. Se afișează cel mai mare număr și cel mai mic număr. Se calculează media aritmetică a numerelor care au valoarea cea mai mare sau au valoarea cea mai mică și se afișează numai numerele care sunt mai mari decât media aritmetică.
18. Pentru două mulțimi de numere A și B să se determine diferențele A-B și B-A.
19. Se citește dintr-un fișier text un număr cu maxim 20 de cifre. Se creează o listă cu cifrele numărului, se elimină din listă cifrele pare și se afișează numărul astfel obținut.
20. Se calculează produsul a două polinoame. Informațiile despre cele două polinoame se citesc dintr-un fișier. (Indicație. Pentru fiecare nod din prima listă se parcurge a doua listă și, dacă produsul coeficientilor nu este nul, se creează un nod în lista a treia care va avea coeficientul egal cu produsul coeficientilor și gradul egal cu suma gradelor. Lista a treia este o listă ordonată descrescător după grad.)
21. Se creează două liste cu n noduri și respectiv m noduri care conțin numere întregi generate aleatoriu în intervalul $[a,b]$. Valorile pentru numărul de noduri n , și respectiv m , și pentru limitele intervalului, a și b , se citesc de la tastatură. Să se afișeze numai numerele distincte din cele două liste și numărul care are cea mai mare frecvență de apariție în cele două liste.

2.6.4. Algoritmi pentru prelucrarea listelor circulare simplu înăntărită

Algoritmii de adăugare a primului nod la lista vidă și de adăugare după ultimul nod sunt la fel ca și cei de la listele simplu înăntărite:



2.6.4.1. Crearea listei

Deoarece în algoritmii de prelucrare trebuie să se cunoască adresa primului nod, este importantă adăugarea primului nod la lista vidă. Pașii algoritmului de creare a unei liste sunt:

PAS1. Se adaugă primul nod la listă (nodul **prim**).

PAS2. Cât timp mai există informație executată: se adaugă un nod la listă după ultimul nod.

PAS3. Se leagă ultimul nod de primul nod.

Implementarea algoritmului. Se folosește funcția procedurală **creare()** al cărei parametru este **prim** de tip **nod**: care se transmite prin referință deoarece este parametru de intrare-iesire. Se folosește variabila globală **n** pentru citirea informației din nod. Se consideră că nu mai există informație atunci când valoarea citită pentru **n** are valoarea 0.

```
void creare(nod *&prim)
{nod *ultim; cin>>n; adauga_nod(prim,ultim);
 while(n!=0) {cin>>n; adauga_ultim(ultim);}
 ultim->urm=prim; }
```

2.6.4.2. Parcurea listei

Deoarece lista circulară nu conține un ultim nod care să fie considerat ca terminator al listei, se va considera ca nod care termină lista nodul **prim**. Pașii algoritmului de prelucrare a unei liste circulare sunt:

- PAS1.** Se prelucrează primul nod din listă (nodul **prim**).
- PAS2.** Pentru fiecare nod din listă începând de la succesorul nodului **prim** până la primul nod, **execuță**: se prelucrează nodul curent.

Implementarea algoritmului. Se folosește funcția procedurală **parcurge()** al cărei parametru **prim** de tip **nod** se transmite prin valoare deoarece este parametru de intrare.

```
void parcurge(nod *prim)
{//se prelucrează prim->info;
for (nod *p=prim->urm; p!=prim; p=p->urm) //se prelucrează p->info;}
```

2.6.4.3. Eliminarea unui nod din listă

Dacă se elimină din listă nodul care urmează după nodul curent **p** trebuie să se verifice dacă acest nod nu este nodul **prim**, ca să nu se piardă adresa primului element din listă.

Pașii algoritmului de eliminare a nodului următor nodului curent sunt:

- PAS1.** Se salvează adresa succesorului nodului **p** în pointerul **q**.
- PAS2.** Se leagă nodul **p** de succesorul succesorului lui.
- PAS3.** Dacă succesorul nodului **p** era nodul **prim**, atunci succesorul nodului **prim** devine nodul **prim**.
- PAS4.** Se cere eliberarea zonei de memorie de la adresa memorată în pointerul **q**.

Implementarea algoritmului. Se folosește funcția procedurală **elimina_urm()** ai cărei parametri de tip **nod** sunt: **p** (pentru adresa nodului precedent nodului ce se elimină), care se transmite prin valoare deoarece este parametru de intrare și **prim**, care se transmite prin referință, deoarece este parametru de intrare-iesire.

```
void elimina_urm(nod *p,nod *&prim)
{nod *q=p->urm; p->urm=p->urm->urm;
 if(q==prim) prim=prim->urm;
 delete q;}
```

Studiu de caz

Scop: exemplificarea modului în care, pentru rezolvarea problemei, folosiți algoritmii de prelucrare a listelor circulare simplu înlățuită și implementarea lor cu ajutorul subprogramelor.

Enunțul problemei. Se citește dintr-un fișier text un sir de numere separate prin spațiu cu care se creează o listă circulară simplu înlățuită în ordinea în care sunt citite numerele din fișier. Să se șteargă numerele pare din listă și să se afișeze numerele din listă. Pentru testarea programului se vor folosi două seturi de numere: {2, 2, 3, 4, 4} și {2, 2, 2, 4, 4}.

Problema se descompune în următoarele subprobleme, iar algoritmii pentru rezolvarea subproblemelor sunt implementați cu ajutorul subprogramelor:

- P1** Se creează lista circulară simplu înlățuită – subprogramul **creare()**.
- P2** Se parcurge lista de la succesorul primului nod până la primul nod și **dacă** numărul din succesorul nodului curent este par, **atunci** se elimină succesorul nodului curent – subprogramul **elimină_urm()**; **altfel** se trece la succesorul nodului curent.
- P3** **Dacă** numărul din nodul **prim** este par, **atunci** se elimină nodul **prim** și lista este vidă – subprogramul **elimină_prim()**.

P4 Dacă lista nu este vidă (subprogramul `este_vida()`), atunci se afișează numerele din fiecare nod (subprogramul `afisare()`).

```
#include<iostream.h>
struct nod {int info;
            nod *urm;};
fstream f("lista7.txt",ios::in);
int x;
void adauga_nod(nod *&prim,nod *&ultim)
{prim=new nod; prim->info=x; prim->urm=NULL; ultim=prim;}
void adauga_ultim(nod *&ultim)
{nod *p=new nod; p->info=x; p->urm=NULL; ultim->urm=p; ultim=p;}
void creare(nod *&prim)
{nod *ultim; f>>x; adauga_nod(prim,ultim);
 while (f>>x) adauga_ultim(ultim);
 ultim->urm=prim;}
void elibera_urm(nod *p,nod *&prim)
{nod *q=p->urm; p->urm=p->urm->urm;
 if(q==prim) prim=prim->urm;
 delete q;}
void elibera_prim(nod *&prim)
{nod *q=prim; prim=NULL; delete q;}
int este_vida(nod *prim) {return prim==NULL;}
void afisare(nod *prim)
{cout<<prim->info<<" ";
 for (nod *p=p->urm;p!=prim;p=p->urm) cout<<p->info<<" "; cout<<endl;}
void main()
{nod *prim,*p; creare(prim); afisare(prim);
 for(p=prim->urm;p!=prim;)
    if (p->urm->info%2==0) elibera_urm(p,prim);
    else p=p->urm;
 if(prim->info%2==0) elibera_prim(prim);
 if (!este_vida(prim)) afisare(prim);}
```


Temă


Scripteți câte un program care să rezolve cerințele fiecărei probleme. Fiecare problemă se va descompune în subprobleme și algoritmul pentru rezolvarea unei subprobleme se va implementa cu un subprogram. Datele se transmit între subprograme cu ajutorul parametrilor de comunicație și nu al variabilelor globale. Se creează **liste circulare simplu înlăntuite** în nodurile cărora se memorează numere întregi. Sirul de numere se citește dintr-un fișier text în care sunt memorate pe același rând, separate prin spațiu. După executarea unei operații de prelucrare a listei, se vor afișa numerele din noduri pentru a verifica dacă operația s-a executat corect. Se vor alege seturi de date de intrare astfel încât să se verifice algoritmul pe toate traseele lui. Pentru următorii 4 itemi liste se creează astfel încât ordinea de acces să fie cea în care sunt citite numerele din fișier.

1. Să se insereze, după fiecare număr divizibil cu cea mai mare cifră a sa, valoarea cifrei, și să se eliminate numerele care au ultimele două cifre consecutive.
2. Să se verifice dacă numerele sunt în progresie geometrică și să se afișeze primul termen al progresiei geometrice. (**Observație.** Dacă numerele sunt în progresie geometrică, nu este obligatoriu ca sirul de numere citit din fișier să înceapă cu primul termen al progresiei geometrice.)
3. Să se insereze între două numere pare din listă media lor aritmetică până când nu mai există perechi de numere pare.

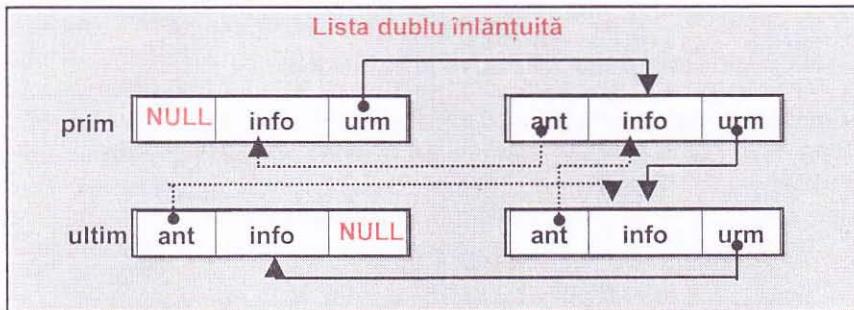
- Din lista circulară creată să se creeze alte două liste circulare simplu înlățuite – una cu numerele divizibile cu cea mai mare cifră, iar alta cu numerele divizibile cu cea mai mică cifră – și să se verifice dacă cele două liste conțin numere comune.
- Se creează o listă ordonată crescător și se creează apoi din această listă o listă cu numerele care sunt pătrate perfecte.

2.6.5. Algoritmi pentru prelucrarea listelor dublu înlățuite

În cazul listelor dublu înlățuite informația de legătură trebuie să conțină și adresa succesorului nodului (pointerul `*ant` către tipul `nod`):

```
struct nod
    {int info;           //informația propriu-zisă
     nod *ant, *urm;}; //informația pentru legătură
     nod *prim, *ultim, *p;
```

Algoritmii de la listele simplu înlățuite se modifică prin adăugarea instrucțiunilor care întrețin și adresa de legătură cu predecesorul nodului. Algoritmul de adăugare a unui nod `p` în interiorul listei înaintea unui nod `q` se simplifică deoarece se cunoaște adresa atât a succesorului, cât și a predecesorului.



2.6.5.1. Adăugarea primului nod la listă

Implementarea algoritmului. Se folosește funcția procedurală `adauga_nod()` ai cărei parametri `prim` și `ultim` de tip `nod` se transmit prin referință deoarece sunt parametri de ieșire.

```
void adauga_nod (nod *&prim, nod *&ultim)
{prim = new nod; prim->info=x; prim->ant=NULL; prim->urm=NULL;
 ultim=prim;}
```

2.6.5.2. Adăugarea unui nod la listă

Adăugare în fața primului nod

Implementarea algoritmului. Se folosește funcția procedurală `adauga_prim()` al cărei parametru `prim` de tip `nod` se transmite prin referință deoarece este parametru de intrare-ieșire.

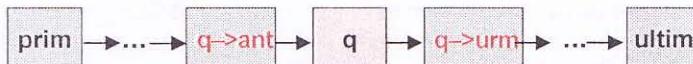
```
void adauga_prim(nod *&prim)
{nod *p=new nod; p->info=x; p->ant=NULL; p->urm=prim; prim=p;}
```

Adăugare după ultimul nod

Implementarea algoritmului. Se folosește funcția procedurală `adauga_ultim()` al cărei parametru `ultim` de tip `nod` se transmite prin referință deoarece este parametru de intrare-ieșire.

```
void adauga_ultim(nod *&ultim)
{nod *p=new nod; p->info=x;
 p->ant=ultim; p->urm=NULL; ultim->urm=p; ultim=p;}
```

Adăugarea în interiorul listei



a) după nodul cu adresa *q*

Nodul *p* care se adaugă se inserează între nodul *q* și nodul *q->urm*. Succesorul său este nodul *q->urm*, iar predecesorul său nodul *q*. Nodul *p* va fi succesorul nodului *q* și predecesorul nodului *q->urm*.

Implementarea algoritmului. Se folosește funcția procedurală `adauga_dupa()` ai cărei parametri sunt de tip *nod*: *q* (adresa nodului după care se face adăugarea), care se transmite prin valoare deoarece este parametru de intrare și *ultim* (adresa ultimului nod), care se transmite prin referință deoarece este parametru de intrare-iesire.

```

void adauga_dupa(nod *q, nod *&ultim)
{nod *p=new nod; p->info=x; p->urm=q->urm; p->ant=q;
if (q==ultim) ultim=p; else q->urm->ant=p; q->urm=p;}
  
```

b) înainte de nodul de adresă *q*

Nodul *p* care se adaugă se inserează între nodul *q->ant* și nodul *q*. Succesorul său este nodul *q*, iar predecesorul său nodul *q->ant*. Nodul *p* va fi succesorul nodului *q->ant* și predecesorul nodului *q*.

Implementarea algoritmului. Se folosește funcția procedurală `adauga_in_fata()` ai cărei parametri sunt de tip *nod*: *q* (adresa nodului înaintea căruia se face adăugarea), care se transmite prin valoare deoarece este parametru de intrare.

```

void adauga_in_fata(nod *q)
{nod *p=new nod; p->info=x;
p->urm=q; p->ant=q->ant; q->ant->urm=p; q->ant=p;}
  
```

2.6.5.3. Parcurea listei

Vizitarea fiecărui nod al listei se poate face în două moduri:

- pornind de la primul nod, până la ultimul nod, în ordinea de înlănțuire a nodurilor – furnizată de adresa *urm* din nodul vizitat;
- pornind de la ultimul nod, până la primul nod, în ordinea de înlănțuire a nodurilor – furnizată de adresa *ant* din nodul vizitat

Implementarea algoritmului. Se folosește funcția procedurală `parcurge_inainte()`, respectiv `parcurge_inapoi()`, al cărei parametru *prim*, respectiv *ultim*, de tip *nod*, se transmite prin valoare deoarece este parametru de intrare.

```

void parcurge_inainte(nod *prim)
{for (nod *p=prim; p!=NULL; p=p->urm) //se prelucrează p->info}
void parcurge_inapoi(nod *ultim)
{for (nod *p=ultim; p!=NULL; p=p->ant) //se prelucrează p->info}
  
```

2.6.5.4. Eliminarea unui nod din listă

Eliminarea primului nod

Implementarea algoritmului. Se folosește funcția procedurală `elimina_prim()` al cărei parametru *prim* de tip *nod* se transmite prin referință deoarece este parametru de intrare-iesire.

```

void elimina_prim(nod *&prim)
{nod *q=prim; prim->urm->ant=NULL; prim=prim->urm; delete q;}
  
```

Eliminarea ultimului nod

Implementarea algoritmului. Se folosește funcția procedurală `elimina_ultim()` al cărei parametru `ultim` de tip `nod` se transmite prin referință deoarece este parametru de intrare-ieșire.

```
void elmina_ultim(nod *&ultim)
{nod *q=ultim; ultim->ant->urm=NULL; ultim=ultim->ant; delete q; }
```

Eliminarea unui nod din interiorul listei

Pentru a elimina nodul `p` aflat în interiorul listei, trebuie să legăm predecesorul nodului `p` (`p->ant`) de succesorul lui (`p->urm`). Succesorul nodului `p->ant` va fi nodul `p->urm`, iar predecesorul nodului `p->urm` va fi nodul `p->ant`.

Implementarea algoritmului. Se folosește funcția procedurală `elmina()` al cărei parametru este de tip `nod`: `p` (adresa nodului care se elimină), care se transmite prin valoare deoarece este parametru de intrare.

```
void elmina(nod *p)
{nod *q=p; p->ant->urm=p->urm; p->urm->ant=p->ant; delete q; }
```

Studiu de caz

Scop: exemplificarea modului în care, pentru rezolvarea problemei, folosiți algoritmi de prelucrare a listelor dublu înlănuite și implementarea lor cu ajutorul subprogramelor.

Enunțul problemei 1. Se citește dintr-un fișier text un sir de numere separate prin spațiu cu care se creează o listă dublu înlănuită în ordinea în care sunt citite numerele din fișier. Să se adauge valoarea 1 după fiecare număr par și să se șteargă apoi numerele pare din listă. Să se afișeze numerele din listă după fiecare operație de prelucrare, în ambele moduri (de la primul la ultimul, și de la ultimul la primul). Pentru testarea programului se va folosi sirul de numere: {2, 2, 5, 3, 4, 4}.

În prelucrarea listelor dublu înlănuite trebuie întreținute atât adresa primului nod, cât și adresa ultimului nod. Problema se descompune în următoarele subprobleme, iar algoritmii pentru rezolvarea subproblemelor sunt implementați cu ajutorul subprogramelor:

P1 Se creează lista dublu înlănuită – subprogramul `creare()`.

P2 Se adaugă un nod cu valoarea 1 după fiecare număr par, astfel (subprogramul `prelucrare_1()`):

- Se parcurge lista de la primul nod până la penultimul nod și, dacă numărul din nodul curent este par, se adaugă după el un nod cu valoarea 1 – subprogramul `adauga_dupa()`.
- Dacă numărul din nodul `ultim` este par, **atunci** se adaugă după el un nod care conține valoarea 1 și acest nod devine nodul `ultim` – subprogramul `adauga_ultim()`.

P3 Se elimină nodurile cu numere pare, astfel (subprogramul `prelucrare_2()`):

- Se parcurge lista de la succesorul primului nod până la penultimul nod și, dacă numărul din nodul curent este par, **atunci** se elimină din listă (subprogramul `elimina_urm()`); **altfel**, se trece la succesorul nodului curent.
- Dacă numărul din nodul `prim` este par, **atunci** se elimină din listă și succesorul său devine nodul `prim` – subprogramul `elimina_prim()`.

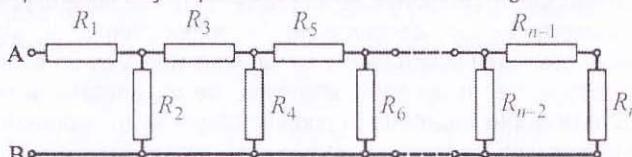
```
#include<iostream.h>
struct nod {int info;
            nod *ant,*urm;};
fstream f("lista8.txt",ios::in);
int x;
```

```

void adauga_nod(nod *&prim, nod *&ultim)
{prim=new nod; prim->info=x;
 prim->urm=NULL; prim->ant=NULL; ultim=prim;}
void adauga_ultim(nod *&ultim)
{nod *p; p=new nod; p->info=x;
 p->urm=NULL; p->ant=ultim; ultim->urm=p; ultim=p;}
void adauga_dupa(nod *p)
{nod *q=new nod; q->info=x;
 q->urm=p->urm; q->ant=p; p->urm->ant=q; p->urm=q;}
void elibera_prim(nod *&prim)
{nod *q=prim; prim->urm->ant=NULL; prim=prim->urm; delete q;}
void elibera(nod *&p)
{nod *q=p; p->ant->urm=p->urm; p->urm->ant=p->ant; p=p->urm;
 delete q;}
void creare(nod *&prim, nod *&ultim)
{f>>x; adauga_nod(prim, ultim);
 while (f>>x) adauga_ultim(ultim);}
void prelucrare_1(nod *&prim, nod *&ultim)
{for (nod *p=prim; p->urm!=NULL; p=p->urm)
    if (p->info%2==0) adauga_dupa(p);
    if (ultim->info%2==0) adauga_ultim(ultim);}
void prelucrare_2(nod *&prim)
{for (nod *p=prim->urm; p->urm!=NULL;)
    if (p->info%2==0) elibera(p); else p=p->urm;
    if (prim->info%2==0) elibera_prim(prim);}
void afisare_urm(nod *&prim)
{for (nod *p=prim; p!=NULL; p=p->urm) cout<<p->info<<" "; cout<<endl;}
void afisare_ant(nod *&ultim)
{for (nod *p=ultim; p!=NULL; p=p->ant) cout<<p->info<<" "; cout<<endl;}
void main()
{nod *prim, *ultim, *p;
 creare(prim, ultim); afisare_urm(prim); afisare_ant(ultim);
 x=1; prelucrare_1(prim, ultim); afisare_urm(prim); afisare_ant(ultim);
 prelucrare_2(prim); afisare_urm(prim); afisare_ant(ultim);}

```

Enunțul problemei 2 – Calcularea rezistenței echivalente. Să se calculeze rezistența echivalentă între punctele A și B pentru circuitul electric din figură.



Pentru calcularea rezistenței echivalente se pornește de la ultimele rezistențe – R_n și R_{n-1} – care sunt legate în serie. Se calculează rezistența lor echivalentă R_{e1}, care va fi legată în paralel cu rezistența R_{n-2}. Prin calcularea rezistenței echivalente a celor două rezistențe legate în paralel, R_{e1} și R_{n-2}, se va obține o nouă rezistență echivalentă R_{e2} care este legată în serie cu rezistența R_{n-3}. Calcularea rezistenței echivalente a circuitului electric este un proces repetitiv în care alternează calcularea unei rezistențe echivalente a două rezistențe legate în serie cu calcularea unei rezistențe echivalente a două rezistențe legate în paralel. Pentru a ști care dintre variantele de calcul se alege, se folosește variabila s care are valoarea 1 dacă rezistențele sunt legate în serie, și valoarea 0 dacă sunt legate în paralel.

Valorile pentru rezistențe se citesc dintr-un fișier text în care sunt memorate pe același rând, separate prin spațiu. Se creează o listă dublu înlățuită în care ordinea de acces este cea în care sunt citite numerele din fișier. Lista se parcurge de la ultimul nod până la primul nod.

```
#include<iostream.h>
struct nod {float info;
            nod *ant, *urm;};
fstream f("rezistente.txt",ios::in);
float x;
void adauga(nod *&prim, nod *&ultim)
{prim=new nod; prim->info=x; prim->urm=NULL; prim->ant=NULL; ultim=prim;}
void adauga (nod *&ultim)
{nod *p; p=new nod; p->info=x;
 p->urm=NULL; p->ant=ultim; ultim->urm=p; ultim=p;}
void creare(nod *&prim, nod *&ultim)
{f>>x; adauga(nod(prim,ultim));
 while (f>>x) adauga(ultim);}
float R(nod *ultim)
{int s=1; float r=ultim->info; nod *p=ultim->ant;
 while (p!=NULL)
 if (s) {r+=p->info; p=p->ant; s=0;}
 else {r=(r*p->info)/(r+p->info); p=p->ant; s=1;}
 return r;}
void main(){nod *prim, *ultim; creare(prim,ultim); f.close();
 cout<<"Rezistenta echivalenta= "<<R(ultim);}
```



Recomandare. Listele dublu înlățuite se folosesc în problemele în care, pentru prelucrarea informațiilor:

- se execută frecvent operații de inserare și de eliminare de noduri;
- lista trebuie parcursă în ambele sensuri.

Exemplu. În problemele în care trebuie prelucrate numere foarte mari (pentru memorarea cărora nu pot fi folosite tipurile de date implementate) se folosesc listele dublu înlățuite:

- pentru operațiile aritmetice (adunare, scădere, înmulțire) listele în care sunt memorate cele două numere vor fi parcuse de la ultimul nod, până la primul nod;
- pentru compararea a două numere sau determinarea numărului de cifre ale numărului, listele în care sunt memorate numerele vor fi parcuse de la primul până la ultimul nod.

Tema Scrieți câte un program care să rezolve cerințele fiecărei probleme. Fiecare problemă se va descompune în subprobleme și algoritmul pentru rezolvarea unei subprobleme se va implementa cu un subprogram. Datele se transmit între subprograme cu ajutorul parametrilor de comunicație și nu al variabilelor globale. Se creează liste dublu înlățuite în nodurile cărora se memorează numere întregi. Sirul de numere se citește dintr-un fișier text în care sunt memorate pe același rând, separate prin spațiu. După executarea unei operații de prelucrare a listei, se vor afișa numerele din noduri pentru a verifica dacă operația s-a executat corect. Se vor alege seturi de date de intrare astfel încât să se verifice algoritmul pe toate traseele lui. Listele se creează astfel încât ordinea de acces să fie cea în care sunt citite numerele din fișier.

1. Să se eliminate numerele prime și să se insereze între fiecare pereche de numere rămase cel mai mare divizor comun al lor.
2. Să se calculeze cifra de control a fiecărui număr și, dacă numărul este divizibil cu cifra de control, cifra este adăugată după număr; altfel, numărul este eliminat din listă. Cifra

- de control a unui număr este suma repetată a cifrelor numărului până când se obține o sumă mai mică decât 10.
3. Să se modifice adresele din nodurile listei astfel încât să se obțină două liste liniare dublu înlănțuite care să conțină numerele din pozițiile pare, respectiv din pozițiile impare.
 4. Se citesc dintr-un fișier două numere foarte mari. Să se scrie următoarele subprograme pentru prelucrarea numerelor:
 - a. calcularea sumei, a diferenței și produsului numerelor;
 - b. compararea a două numere (subprogramul trebuie să verifice dacă cele două numere sunt egale, iar dacă nu sunt egale, să precizeze care număr este mai mare);
 - c. verificarea numărului dacă are un număr par sau un număr impar de cifre, fără să se numere cifrele numărului;
 - d. determinarea numărului de cifre ale unui număr;
 - e. verificarea numărului dacă este palindrom.

2.6.6. Algoritmi pentru prelucrarea stivelor

Cele două extremități ale stivei se numesc **vârf** și **bază**. Accesul la nodurile stivei (adăugarea, extragerea sau consultarea unui nod) este permis numai printr-o singură extremitate numită **vârf** și ultimul nod inserat este primul nod extras (se extrage cea mai nouă informație adăugată). La o operație de adăugare a unui nod, vârful stivei stivei urcă, iar la o operație de extragere a unui nod, vârful stivei coboară.

Implementarea dinamică a stivei se face la fel ca a unei liste liniare, cu deosebirea că:

- vârful stivei va fi indicat de pointerul **varf** către tipul **nod**;
- pentru adăugarea unui nod, se poate folosi numai **algoritmul de adăugare în fața primului nod**;
- pentru extragerea unui nod se poate folosi numai **algoritmul pentru eliminarea primului nod**.

Altfel spus, prelucrarea unei stive se face de la **vârf spre bază** și se prelucreză întotdeauna nodul din **vârful stivei**. Accesul la informația din acest nod se face cu **varf->info**.

Stiva vidă este stiva care nu conține nici un nod și adresa nodului **vârf** are valoarea **NULL** (**varf = NULL;**). În stiva vidă nu se mai pot executa operații de extragere de noduri. Stiva poate să ajungă să fie vidă în două cazuri: la **initializare** sau după **extragerea ultimului nod**. Pentru testarea unei stive dacă este vidă, se poate implementa funcția operand **este_vida()** care va furniza valoarea 1 („adevărat”), dacă stiva este vidă, și valoarea 0 („fals”) dacă stiva nu este vidă.

```
int este_vida(nod *varf)
{return varf==NULL; }
```

Pentru prelucrări cu ajutorul stivei puteți folosi următorii algoritmi:

- **initializarea stivei**;
- **adăugarea unui nod la stivă**;
- **extragerea unui nod din stivă**;
- **consultarea nodului din vârful stivei**.

2.6.6.1. Initializarea stivei

Prin acest algoritm se creează stiva vidă. În acest caz, nodul **vârf** nu există și pointerul **varf** are valoarea **NULL**.

Implementarea algoritmului. Se folosește funcția procedurală **init()** al cărei parametru **varf** se transmite prin referință, deoarece este parametru de ieșire.

```
void init(nod *&varf)
{varf = NULL;}
```

2.6.6.2. Adăugarea unui nod la stivă

Nodul se adaugă la stivă ca predecesor al vârfului. Pașii algoritmului sunt:

PAS1. Se cere alocarea de memorie pentru nodul p.

PAS2. Se scrie informația în nodul p.

PAS3. Nodul p se leagă de nodul varf.

PAS4. Nodul p adăugat devine nodul varf.

Implementarea algoritmului. Se folosește funcția procedurală **adauga()** al cărei parametru varf se transmite prin referință, deoarece este parametru de intrare-iesire.

```
void adauga(nod *&varf)
{nod *p=new nod; p->info=x; p->urm=varf; varf=p;}
```

2.6.6.3. Extragerea unui nod din stivă

Nodurile se extrag din stivă pentru a putea consulta informația care există în nodurile următoare. Un nod se poate extrage numai în cazul în care stiva nu este vidă. Se poate extrage numai nodul din vârful stivei (se eliberează spațiul care a fost ocupat de nod). În vârful stivei va ajunge succesorul nodului extras. Pașii algoritmului sunt:

PAS1. Se salvează adresa nodului varf în pointerul p.

PAS2. Succesorul nodului varf devine nodul varf.

PAS3. Se cere eliberarea zonei de memorie de la adresa memorată în pointerul p.

Implementarea algoritmului. Se folosește funcția procedurală **extrage()** al cărei parametru varf se transmite prin referință, deoarece este parametru de intrare-iesire.

```
void extrage (nod *&varf)
{nod *p =varf; varf=varf->urm; delete p;}
```

2.6.6.4. Prelucrarea stivei

Prin acest algoritm se consultă informația din fiecare nod al stivei. Deoarece nu poate fi consultată decât informația din vârful stivei, pentru a ajunge la un nod trebuie să se extragă toate nodurile până la el.

Implementarea algoritmului. Se folosește funcția procedurală **prelucrare()** al cărei parametru varf de tip nod se transmite prin referință, deoarece este parametru de intrare-iesire.

```
void prelucrare(nod *&varf)
{while (varf!=NULL) { //se prelucrează varf->info
    extrage(varf);}}
```

Studiu de caz

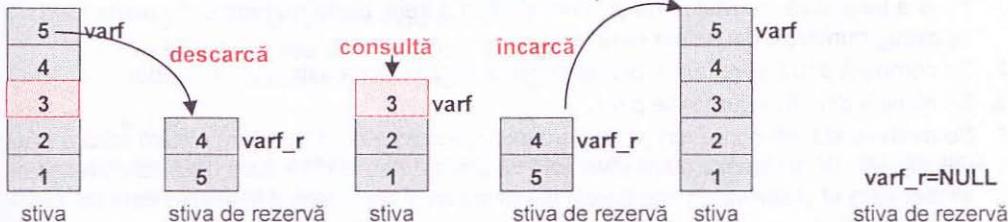
Scop: exemplificarea modului în care, pentru rezolvarea problemei, folosiți algoritmii de prelucrare a stivelor și implementarea lor cu ajutorul subprogramelor.

Enunțul problemei. Se citește dintr-un fișier text un sir de numere separate prin spațiu care se depun într-o stivă în ordinea în care sunt citite numerele din fișier. Să se eliminate numărul de la baza stivei și numerele rămase în stivă să se scrie într-un fișier text. Pentru testarea programului se va folosi sirul de numere: {1, 2, 5, 3, 4}.

În prelucrarea stivelor, pentru a putea ajunge la informația care trebuie prelucrată, trebuie extrase toate nodurile, până la nodul care conține acea informație, deoarece nu poate fi prelucrată decât informația din vârful stivei. Pentru a nu se pierde informația din nodurile

extrase, ele vor fi descărcate într-o altă stivă (de rezervă), iar după prelucrarea informației, nodurile vor fi încărcate din nou în stivă.

Consultarea nodului care conține numărul 3



Problema se descompune în următoarele subprobleme, iar algoritmul de rezolvare a unei subprobleme este implementat cu ajutorul unui subprogram:

- P1 Se creează stiva cu numerele citite din fișier (nodul **varf**) – subprogramul **creare()**.
- P2 Se descarcă stiva într-o altă stivă (nodul **varf_r**) până la penultimul nod (subprogramul **descarca()**).
- P3 Se extrage ultimul nod din stivă – subprogramul **extrage()**.
- P4 Se încarcă în stivă (nodul **varf**) nodurile din stiva de rezervă (nodul **varf_r**) – subprogramul **incarca()**.
- P5 Se scriu numerele din stivă (nodul **varf**) în fișierul text – subprogramul **salveaza()**.

```
#include<iostream.h>
struct nod {int info;
            nod *urm;};
fstream f1("stival.txt",ios::in),f2("stiva2.txt",ios::out);
int x;
void init(nod *&varf) {varf=NULL;}
void adauga(nod *&varf) {nod *p=new nod; p->info=x; p->urm=varf; varf=p;}
int este_vida(nod *&varf) {return varf==NULL;}
void extrage(nod *&varf) {nod *p=varf; varf=varf->urm; delete p;}
void creare(nod *&varf)
{init(varf); while (f1>>x) adauga(varf);}
void descarca(nod *&varf,nod *&varf_r)
{init(varf_r);
while (varf->urm!=NULL) {x=varf->info; extrage(varf); adauga(varf_r);}
void incarca(nod *&varf,nod *&varf_r)
{while (!este_vida(varf_r))
 {x=varf_r->info; extrage(varf_r); adauga(varf);}
void salveaza(nod *&varf)
{while (!este_vida(varf)) {f2<<varf->info<<" "; extrage(varf);}
void main() {nod *varf,*varf_r; creare(varf); f1.close();
            descarca(varf,varf_r); extrage(varf);
            incarca(varf,varf_r); salveaza(varf); f2.close();}
```



Tema



Fiecare problemă se va descompune în subprobleme și algoritmul pentru rezolvarea unei subprobleme se va implementa cu un subprogram. Datele se transmit între subprograme cu ajutorul parametrilor de comunicație și nu al variabilelor globale. Se creează **stive** în nodurile cărora se memorează informația (numere sau caractere). Informația se citește dintr-un fișier text în care, numerele sau caracterele, sunt memorate pe același rând, separate prin spațiu. Se vor alege seturi de date de intrare astfel încât să se verifice algoritmul pe toate traseele lui.

1. Într-o stivă sunt memorate numere din intervalul [1,10], ordonate crescător, iar într-o altă stivă, numere din intervalul [20,30], ordonate crescător. Să se concateneze cele două siruri de numere, păstrând ordonarea crescătoare. (Indicație. Se răstoarnă a doua stivă într-o a treia stivă, se răstoarnă și prima stivă în a treia, peste numerele din prima stivă, și se extrag numerele din a treia stivă.)
2. Se compară două stive fără a pierde conținutul lor în urma extragerii de noduri.
3. Se elimină din stivă numerele pare.
4. Se memorează într-o stivă un sir de litere mici. Se citește de la tastatură o literă mică a alfabetului – lit. Să se creeze două stive: una va conține literele din stiva inițială care preced în alfabet litera lit și alta va conține literele din stiva inițială care succed în alfabet litera lit.
5. Să se afișeze în ordine inversă numerele dintr-o listă liniară simplu înlățuită. (Indicație. Se încarcă numerele din listă într-o stivă, în ordinea de parcursare a listei, și apoi se extrag și se afișează numerele din stivă.)
6. Se ordonează crescător un sir de numere, cu ajutorul a două stive. Determinați complexitatea algoritmului. (Indicație. Se folosesc două stive. Se scrie un număr în Stiva 1. Se citește un număr. Dacă numărul din vârful Stivei 1 este mai mare decât numărul citit, atunci noul număr se adaugă la Stiva 1; altfel, se descarcă din Stiva 1 în Stiva 2 numere până când în vârful Stivei 1 ajunge un număr mai mare decât numărul citit, se adaugă la Stiva 1 numărul citit, și se încarcă în Stiva 1 numerele din Stiva 2.)
7. Să se verifice dacă o expresie aritmetică ce conține paranteze este balansată, adică dacă fiecare paranteză deschisă este închisă corect. De exemplu, expresia $(a+b+\{c/[d-e]\})+(d/s)$ este balansată, iar expresia $(a+b+\{c/[d-e]\})+(d/s)$ nu este balansată. (Indicație. Se codifică parantezele 1–(; 2–[; 3–{; 4–}; 5–]; 6–}). Dacă stiva nu este vidă și dacă diferența dintre codul dintre paranteza citită și codul parantezei din vârful stivei este 3, atunci se elimină nodul din vârful stivei; altfel, se adaugă la stivă codul parantezei citite. Dacă stiva este vidă la terminarea evaluării expresiei, înseamnă că expresia este balansată.)

2.6.7. Algoritmi pentru prelucrarea cozilor

Cele două extremități ale cozii se numesc **cap** și **bază**. Adăugarea de noduri la coadă se face prin nodul **bază**, iar extragerea și consultarea unui nod este permisă numai prin extremitatea **cap** (se extrage cea mai veche informație adăugată).

Implementarea dinamică a cozii se face la fel ca a unei liste liniare, cu deosebirea că:

- primul nod al cozii va fi indicat de pointerul **cap** către tipul **nod**, iar ultimul nod al cozii va fi indicat de pointerul **baza** către tipul **nod**;
- pentru adăugarea unui nod, se poate folosi numai **algoritmul de adăugare după ultimul nod**;
- pentru extragerea unui nod se poate folosi numai **algoritmul pentru eliminarea primului nod**.

Altfel spus, prelucrarea unei cozi se face **de la cap spre bază** și se prelucreză întotdeauna nodul din **capul cozii**. Accesul la informația din acest nod se face cu **cap->info**.

Coada vidă este coada care nu conține nici un nod și adresa nodului **cap** are valoarea **NULL** (**cap = NULL;**). În coada vidă nu se mai pot executa operații de extragere de noduri. Coada poate să ajungă să fie vidă în două cazuri: la **initializare** sau după **extragerea ultimului nod**. Pentru testarea unei cozi dacă este vidă se poate implementa funcția operand **este_vidă()** care va furniza valoarea 1 („adevărat”), dacă este vidă, și valoarea 0 („fals”) dacă nu este vidă.

```
int este_vida(nod *cap)
{return cap==NULL; }
```

Pentru prelucrări cu ajutorul cozii puteți folosi următorii algoritmi:

- inițializarea cozii;
- adăugarea unui nod la coadă;
- extragerea unui nod din coadă;
- consultarea nodului din capul cozii.

2.6.7.1. Inițializarea cozii

Prin acest algoritm se creează coada care conține un nod. În acest caz, nodurile cap și bază vor avea aceeași adresă.

Implementarea algoritmului. Se folosește funcția procedurală `init()` al cărei parametri cap și baza se transmit prin referință, deoarece sunt parametri de ieșire.

```
void init(nod *&cap, nod *&baza)
{cap=new nod; cap->info=x; cap->urm=NULL; baza=cap; }
```

2.6.7.2. Adăugarea unui nod la coadă

Nodul se adaugă la coadă ca succesor al bazei. Pașii algoritmului sunt:

- PAS1. Se cere alocarea de memorie pentru nodul p.
- PAS2. Se scrie informația în nodul p.
- PAS3. Nodul p se leagă de nodul baza.
- PAS4. Nodul p adăugat devine nodul baza.

Implementarea algoritmului. Se folosește funcția procedurală `adauga()` al cărei parametru baza se transmite prin referință deoarece este parametru de intrare-ieșire.

```
void adauga(nod *&baza)
{nod *p=new nod; p->info=x; p->urm=NULL; baza->urm=p; baza=p; }
```

2.6.7.3. Extragerea unui nod din coadă

Nodurile se extrag din coadă pentru a putea consulta informația care există în nodurile următoare. Un nod se poate extrage numai în cazul în care coada nu este vidă. Se poate extrage numai nodul din capul cozii (se eliberează spațiul care a fost ocupat de nod). În capul cozii va ajunge succesorul nodului extras. Pașii algoritmului sunt:

- PAS1. Se salvează adresa nodului cap în pointerul p.
- PAS2. Succesorul nodului cap devine nodul cap.
- PAS3. Se cere eliberarea zonei de memorie de la adresa memorată în pointerul p.

Implementarea algoritmului. Se folosește funcția procedurală `extrage()` al cărei parametru cap se transmite prin referință, deoarece este parametru de intrare-ieșire.

```
void extrage(nod *&cap)
{nod *p =cap; cap=cap->urm; delete p; }
```

2.6.7.4. Prelucrarea cozii

Prin acest algoritm se consultă informația din fiecare nod al cozii. Deoarece nu poate fi consultată decât informația din capul cozii, pentru a ajunge la un nod trebuie să extragă toate nodurile până la el.

Implementarea algoritmului. Se folosește funcția procedurală `prelucrare()` al cărei parametru cap de tip nod se transmite prin referință, deoarece este parametru de intrare-ieșire.

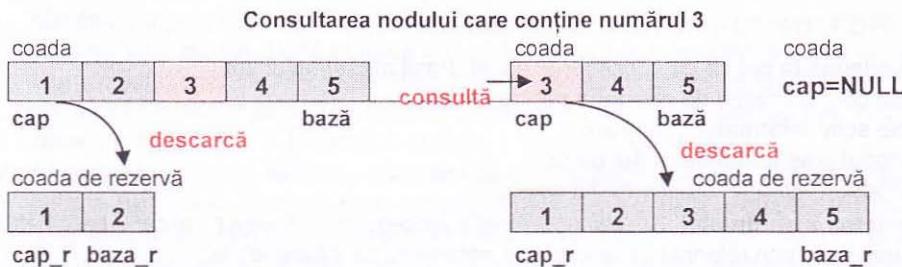
```
void prelucrare(nod *&cap)
{while (cap!=NULL) { //se prelucrează cap->info
extrage(cap); }}
```

Studiu de caz

Scop: exemplificarea modului în care, pentru rezolvarea problemei, folosiți algoritmii de prelucrare a cozilor și implementarea lor cu ajutorul subprogramelor.

Enunțul problemei. Se citește dintr-un fișier text un sir de numere separate prin spațiu care se depun într-o coadă în ordinea în care sunt citite din fișier. Să se eliminate numărul din mijlocul cozii, dacă numărul de noduri este impar, și cele două numere din mijloc, dacă numărul de noduri este par, iar numerele rămase în coadă să se scrie într-un fișier text. Pentru testarea programului se vor folosi două seturi de numere: {1, 2, 5, 3, 4} și {1, 2, 5, 4}.

În prelucrarea cozilor, pentru a putea ajunge la informația care trebuie prelucrată, trebuie extrase toate nodurile până la nodul care conține acea informație, deoarece nu poate fi prelucrată decât informația din capul cozii. Pentru a nu se pierde informația din nodurile extrase, ele vor fi descărcate într-o altă coadă (de rezervă), iar după prelucrarea informației, nodurile vor fi descărcate în continuare în coada de rezervă, până când coada inițială devine vidă.



Problema se descompune în următoarele subprobleme, iar algoritmul de rezolvare a unei subprobleme este implementat cu ajutorul unui subprogram:

- P1 Se creează coada cu numerele citite din fișier (nodurile **cap** și **baza**) – subprogramul **creare()**.
- P2 Se descarcă coada într-o altă coadă (nodurile **cap_r** și **baza_r**) până la nodurile care trebuie eliminate – subprogramul **descarca_1()**.
- P3 Se extrage nodul sau se extrag nodurile din mijlocul cozii – subprogramul **extrage()**.
- P4 Se descarcă restul nodurilor din coadă (nodul **cap**) în coada de rezervă (nodul **baza_r**) – subprogramul **descarca_2()**.
- P5 Se scriu numerele din coada de rezervă (nodul **cap_r**) în fișierul text – subprogramul **salveaza()**.

```
#include<iostream.h>
struct nod {int info;
            nod *urm;};
fstream f1("coada1.txt",ios::in),f2("coada2.txt",ios::out);
int x,n,i,j;
void init(nod *&cap,nod *&baza)
{cap=new nod; cap->info=x; cap->urm=NULL; baza=cap;}
int este_vida(nod *cap) {return cap==NULL;}
void adauga(nod *&baza)
{nod *p=new nod; p->info=x; p->urm=NULL; baza->urm=p; baza=p;}
void extrage(nod *&cap) {nod *p=cap; cap=cap->urm; delete p;}
void creare(nod *&cap, nod *&baza)
{f1>>x; init(cap,baza); n++;
```

```

while (f1>>x) {adauga(baza); n++;}
void descarca_1(nod *&cap,nod *&cap_r, nod *&baza_r)
{x=cap->info; extrage(cap); init(cap_r,baza_r);i++;
 while (i<j) {x=cap->info; extrage(cap); adauga(baza_r);i++;}}
void descarca_2(nod *&cap,nod *&baza_r)
{while (!este vida(cap))
 {x=cap->info; extrage(cap); adauga(baza_r);}}
void salveaza(nod *&cap)
{while (!este vida(cap)) {f2<<cap->info<<" "; extrage(cap);}}
void main()
{nod *cap,*baza,*cap_r,*baza_r; creare(cap,baza); f1.close();
if (n%2==0) j=n/2-1; else j=n/2;
descarca_1(cap, cap_r, baza_r); extrage(cap);
if (n%2==0) extrage(cap);
descarca_2(cap, baza_r); salveaza(cap_r); f2.close();}

```


Tema


Fiecare problemă se va descompune în subprobleme și algoritmul pentru rezolvarea unei subprobleme se va implementa cu un subprogram. Datele se transmit între subprograme cu ajutorul parametrilor de comunicație și nu al variabilelor globale. Se creează cozi în nodurile cărora se memoră numere întregi. Sirul de numere se citește dintr-un fișier text în care sunt memorate pe același rând, separate prin spațiu. Se vor alege seturi de date de intrare astfel încât să se verifice algoritmul pe toate traseele lui.

1. Se elimină din coadă numerele pare.
2. Se verifică dacă numerele dintr-o coadă sunt ordonate (crescător sau descrescător).
3. Se concatenează două cozi, adăugând a doua coadă la sfârșitul primei cozi.
4. Se formează din două cozi o a treia coadă care conține mai întâi numerele pare din pozițiile impare din prima coadă și apoi numerele impare din pozițiile pare din a doua coadă.
5. Se inversează ordinea numerelor dintr-o coadă cu ajutorul unei stive.

2.6.8. Aplicații practice

1. **Analiza lexicală a unui text.** Să se afișeze, în ordine alfabetică, cuvintele dintr-un text și frecvența lor de apariție în text.
2. Există două automate care elibereză bonuri de ordine pentru o coadă de așteptare. Pe fiecare bon de ordine este trecut numărul bonului și momentul la care a fost eliberat (exprimat în oră, minut și secundă). Bonurile au numere unice. Deservirea persoanelor se face în ordinea momentului de eliberare a bonurilor. Să se organizeze o coadă de așteptare pe baza bonurilor emise de cele două automate.
3. **Jocul lui Josephus.** Într-un grup de **n** copii, aceștia sunt aranjați în cerc și sunt numărați începând cu primul copil până la numărul **k**. Copilul care are numărul **k** ieșe din joc, iar numărătoarea începe din nou de la 1 cu următorul copil. Să se afișeze ordinea de ieșire din joc a copiilor. Se vor implementa două soluții, folosind o structură de date de tip:
 - a) coadă;
 - b) listă circulară simplu înlănită.
4. Să se simuleze, cu ajutorul unei stive, o mașină de adunat și multiplicat. Numerele se introduc într-o stivă – și operația se încheie atunci când se citește de la tastatură operatorul „+” (pentru adunarea numerelor) sau operatorul „*” (pentru înmulțirea numerelor). Se mai folosesc: caracterul „A”, pentru a anula ultimul număr introdus, și caracterul „C”, pentru a anula toate numerele introduse.

5. Un automobil trebuie să parcurgă un traseu care formează un poligon, cu întoarcere la stația de pornire. Pe traseu există n stații de alimentare cu carburant care formează vârfurile poligonului. Fiecare stație de alimentare i este caracterizată de coordonatele (x_i, y_i) . Automobilul consumă 1 litru de carburant la fiecare 20 km și nu există restricție pentru capacitatea rezervorului. Se citesc dintr-un fișier text următoarele informații: de pe primul rând numărul de stații n , de pe următorul rând un sir de n numere c_i care reprezintă cantitatea de combustibil cu care este alimentat la stația i , iar de pe următorul rând n perechi de numere x_i și y_i care reprezintă coordonatele stației i . Din ce punct trebuie să plece automobilul astfel încât să parcurgă traseul cu întoarcere în punctul de pornire și să nu rămână fără combustibil.

Evaluare

Pentru exercițiile următoare, dacă nu se specifică semnificația variabilelor de memorie, se folosesc următoarele date:

```
struct nod {int info;
            nod *urm;};
nod *prim,*ultim,*p,*q,*r,*u,*vf,*cap,*baza;
int x,k;
```

Răspundetii:

1. Analizați din punct de vedere al complexității algoritmi pentru prelucrarea listelor. Se va folosi în determinarea complexității timpul maxim de execuție. Pentru compararea algoritmilor completați următorul tabel:

	Lista simplu înăntărită		Lista dublu înăntărită	
	Neordonată	Ordonată	Neordonată	Ordonată
Caută (L, k)				
Adaugă (L, x)				
Elimină (L, x)				
Succesorul (L, x)				
Predecesorul (L, x)				
Minim (L)				
Maxim (L)				

2. O listă simplu înăntărită conține, în ordine, următoarele noduri: $2 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 10 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 18 \rightarrow 20$. Ce se va afișa în urma execuției următoarei secvențe de program?

```
for (p=prim, k=1; k<5; k++, p=p->urm); cout<< p->info;
```

3. O listă simplu înăntărită conține, în ordine, următoarele noduri: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$. Ce se va afișa în urma execuției următoarei secvențe de program?

```
for (k=0, p=prim; p->urm!=NULL; p=p->urm) if (p->info%2) k+= p->info;
cout<<k;
```

4. Ce se va afișa în urma execuției următoarei secvențe de program, dacă lista simplu înăntărită conține, în ordine, următoarele noduri: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$? Dar dacă nodurile ei sunt, în ordine: $1 \rightarrow 0 \rightarrow 3 \rightarrow 0 \rightarrow 5$?

```
for (p=prim; p->info!=0; p=p->urm); cout<<p->info;
```

5. Ce se va afișa în urma execuției următoarei secvențe de program, dacă lista simplu înăntărită conține, în ordine, următoarele noduri: $1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 9$? Dar dacă nodurile ei sunt, în ordine: $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8$?

```
for (k=0, p=prim; p->info!=0; p=p->urm)
```

```

if (p->urm->info==p->info) k++;
cout<<k;

```

Adevărat sau Fals:

- Dacă **p** este primul nod al listei, pentru a afișa informația din al doilea nod se execută secvența de instrucțiuni: **p=p->urm; cout<<p->info;**
- Dacă **p** este primul nod al listei, pentru a afișa informația din al doilea nod se execută instrucțiunea: **cout<<p->urm->info;**
- Dacă **p** este primul nod al listei, pentru a afișa informația din al treilea nod se execută instrucțiunea: **cout<<p->urm->urm->info;**

Alegeți:

- Dacă **p** este primul nod al listei, iar **q** al doilea nod al listei, prin ce instrucțiune se leagă nodul **p** de nodul **q**?
 - p=q;**
 - q=p->urm;**
 - p=q->urm;**
 - p->urm=q;**
- Care dintre următoarele variante realizează corect legăturile în cazul inserării unui nod nou într-o listă simplu înlănțuită, dacă nodul nou are adresa **p**, iar nodul după care se inserează are adresa **q**?
 - p->urm=q; q->urm=p->urm;**
 - p->urm=q->urm; q->urm=p;**
 - q->urm=p->urm;**
 - p->urm=q->urm; p->urm=q;**
- Dacă **p** este primul nod al listei, ce instrucțiune trebuie executată pentru a afișa informația memorată în al treilea nod?
 - cout<<p->urm->urm->info->info;**
 - cout<<p->urm->info->urm->info;**
 - cout<<p->urm->urm->info;**
 - cout<< p->urm->urm->urm->info;**
- Dacă **p**, **q** și **r** sunt trei noduri consecutive ale listei, pentru a interschimba nodul **q** cu nodul **r**, care dintre secvențele de instrucțiuni este corectă?
 - r->urm=q; q->urm=r->urm; p->urm=r;**
 - p->urm=r; r->urm=q; q->urm=r->urm;**
 - q->urm=r->urm; r->urm=q; p->urm=r;**
 - r->urm=q; p->urm=r; q->urm=r->urm;**
- Dacă **p** este un nod al listei și **q** un pointer către tipul nod, care dintre secvențele de instrucțiuni următoare realizează corect legăturile astfel încât să se eliminate din listă cele două noduri care urmează după nodul **p**?
 - p->urm=p->urm->urm->urm;**
 - p->urm=p->urm->urm;**
 - p->urm->urm=p->urm->urm;**
 - q=p->urm; p->urm=q->urm->urm;**
- Dacă **L1** este o listă organizată ca stivă, stabiliți care este adresa corectă de extragere din stivă:
 - delete p; p=p->urm;**
 - r=p->urm; p=r; delete p;**
 - r=p; p=p->urm; delete p;**
 - r=p->urm; p=r; delete r;**
- Care dintre următoarele secvențe de program calculează, în variabila **k**, suma elementelor din lista simplu înlănțuită:
 - for(k=0,p=prim;p!=NULL;p=p->urm) k+= p->info;**
 - for(k=0,p=prim; p->urm!=NULL;p= p->urm) k+=p->info;**
 - k=0; p=prim; while (p!=NULL) {k+= p->info; p= p->urm; }**
 - p=prim; k= p->info;
 do {p= p->urm; k+= p->info;} while (p->urm!=NULL);**

În următorii 7 itemi, variabilele **p** și **q** memorează adresele de început ale listelor liniare simplu înlănțuite nevide **L1** și respectiv **L2**. Elementele listelor sunt de tipul **nod**.

(Bacalaureat – Sesiunea iunie-iulie 2003)

8. Trebuie mutat primul element al listei L1 imediat după primul element al listei L2, în rest listele rămânând neschimbate. Care dintre următoarele atribuiri sunt necesare și în ce ordine se efectuează? 1) $r=q->urm$; 2) $r=p->urm$; 3) $q->urm=p$; 4) $p=r$; 5) $p->urm=r$; 6) $p->urm =q->urm$;
 a. 1 6 3 4 b. 1 3 5 c. 2 6 3 4 d. 2 3 6 4
9. Trebuie mutat primul element al listei L1 imediat după primul element al listei L2, în rest listele rămânând neschimbate. Care dintre următoarele atribuiri sunt necesare și în ce ordine se efectuează? 1) $r=q->urm$; 2) $r=p->urm$; 3) $q->urm=p$; 4) $p=r$; 5) $p->urm=r$; 6) $p->urm =q->urm$;
 a. 1 6 3 4 b. 1 3 5 c. 2 6 3 4 d. 2 3 6 4
10. Dacă la sfârșitul executării secvenței alăturate valoarea variabilei r este nulă, atunci lista L1:
 a. are cel puțin două elemente b. este vidă
 c. este incorrect constituită d. nu este circulară $r=p->urm;$
 $\text{while } (r!=p \& r)$
 $\quad \quad \quad r=r->urm;$
11. Funcția egale(ad1,ad2) returnează valoarea 1 dacă și numai dacă informațiile utile memorate la adresele ad1 și ad2 coincid, altfel returnează valoarea 0. Secvența alăturată calculează în variabila întreagă n numărul de elemente din lista L1:
 a. distincte consecutive aflate la începutul listei
 b. egale consecutive aflate la începutul listei
 c. care sunt egale cu primul element
 d. care sunt egale două câte două $n=0; r=p;$
 $\text{while } (\text{egale}(r,p) \& r)$
 $\quad \quad \quad \{r=r->urm; n++; \}$
12. Dacă L1 este o coadă, cu p adresa primului element și u adresa ultimului element, iar r este adresa unui element ce urmează a fi adăugat în coadă, stabiliți care dintre următoarele este o operație corectă de adăugare:
 a. $r=u->urm; u=r;$ b. $r->urm=p; p=r;$
 c. $u->urm=r; u=r;$ d. $r->urm=u; u=r;$
13. Pentru a uni listele L1 și L2 plasând lista L1 în continuarea listei L2, se efectuează operațiile:
 a. $r=q; \text{while } (r->urm) \{r=r->urm; r->urm=p;\}$
 b. $r=p; \text{while } (r) \ r=r->urm; r=q;$
 c. $r=q; \text{while } (r->urm) \ r=r->urm; r->urm=p;$
 d. $r=p; \text{while } (r->urm) \ r=r->urm; r->urm=q;$
14. Pentru a determina numărul de elemente ale listei L1 se utilizează o variabilă întreagă n astfel:
 a. $n=0; r=p; \text{while } (r->urm) \{r=r->urm; n++; \}$
 b. $n=0; r=p; \text{while } (r) \ r=r->urm; n++;$
 c. $n=0; r=p; \text{while } (r) \ \{r=r->urm; n++; \}$
 d. $n=0; r=p; \text{do } \{n++; \} \text{ while } (r)$

În următorii 6 itemi, variabilele p și u memorează adresa primului, respectiv a ultimului element al listei liniare simplu înlăntuite nevide L. Elementele listelor sunt de tipul nod.

(Bacalaureat – Sesiunea august 2003)

15. Știind că L este formată din 4 elemente, atunci adresa penultimului element este:
 a. $p->urm->urm->$ b. $p->urm->urm$ c. $p->urm$ d. $p->urm->urm->urm$
16. Elementele din lista L aflate la adresele q și r sunt vecine (consecutive) în listă dacă și numai dacă:
 a. $q->urm == r->urm$ b. $r->urm == q \&& q->urm == r$
 c. $q->urm == r \mid\mid r->urm == q$ d. $q == r$

17. Știind că este definită o funcție **cnt** astfel încât **cnt(a1,a2)** returnează numărul de elemente situate în listă între elementele de la adresa **a1** și **a2** (fără a număra elementele de la adresele respective), care dintre următoarele expresii arată al cătelea este elementul memorat la adresa **q** în lista L?

- a. **cnt (p, q)+2** b. **cnt (q, u)+2** c. **cnt (q, u)+1** d. **cnt (p, q)+1**

18. Este definită o funcție **min** astfel încât **min(a1,a2)** returnează valoarea 1 dacă și numai dacă cel puțin unul dintre elementele memorate la adresele **a1** și **a2** se află în lista L și returnează valoarea 0 în caz contrar. Care dintre următoarele expresii are valoarea 1 dacă și numai dacă elementul de la adresa **q** se află în lista L?

- a. **min(p, q)** b. **min(q, u)** c. **min(p, u)** d. **min(q, q)**

19. Un element aflat la adresa **q** face parte din lista L dacă la sfârșitul executării secvenței alăturate variabila **r** are valoarea:

- a. 1 b. p c. q d. **NULL / 0**

```
r=p;
while (r!=q && r)
    r=r->urm;
```

20. Lista L are exact două elemente dacă:

- | | |
|------------------------|--|
| a. p->urm==u | b. u->urm==NULL / !u->urm |
| c. p=u | d. p->urm==NULL / !p->urm |

21. Dacă în variabila **p** este memorată inițial adresa primului nod al unei liste simplu înlăntuită cu cel puțin 10 elemente, pentru a obține în variabila **p** adresa penultimului nod al listei se execută secvența:

- | | |
|--|---|
| a. while (!p) p->urm; | b. while (!p->urm) p->urm; |
| c. while (p->urm->urm) p->urm; | d. while (p->urm) p->urm; |

(Bacalaureat – Sesiunea specială 2003)

22. Știind că într-o listă circulară simplu înlăntuită cu cel puțin două elemente, adresele **p** și **q** reprezintă adresele a două elemente distincte din listă, atunci elementul memorat la adresa **p** este succesorul elementului memorat la adresa **q** în listă dacă și numai dacă:

- | | |
|-----------------------------------|----------------------------------|
| a. p->urm == q; | b. q->urm == p |
| c. p->urm == q->urm; | d. q->urm->urm == p |

(Bacalaureat – Sesiunea specială 2004)

23. Dacă într-o listă circulară simplu înlăntuită, cu cel puțin 4 elemente, se cunoaște adresa **p** a unui element din listă, atunci este accesibilă adresa elementului din listă precedent celui aflat la adresa **p**?

- a. Nu.
 b. Da, în orice situație.
 c. Da, numai dacă **p** este adresa primului element al listei.
 d. Da, numai dacă **p** este adresa ultimului element al listei.

(Bacalaureat – Sesiunea iunie-iulie 2004)

24. Într-o listă liniară simplu înlăntuită nevidă, pentru eliminarea elementului ce urmează după elementul aflat la adresa **p** (elementul de la adresa **p** nu este nici primul, nici ultimul) un elev utilizează trei instrucțiuni simple (nestructurate). Care dintre instrucțiunile următoare poate fi una dintre cele trei?

- | | |
|--|-------------------------|
| a. p->urm->urm = p; | b. dispose (p) |
| c. p->urm = p->urm->urm; | d. p->urm = p |

(Bacalaureat – Sesiunea iunie-iulie 2004)

25. Nodurile unei liste dublu înlăntuite rețin în câmpurile **info**, **adp** și **adu** o informație numerică, adresa nodului precedent și respectiv adresa nodului următor. Știind că lista este corect construită și că două noduri **p** și **q** ale acesteia se învecinează, atunci:

- a. **p->adp==q->adu** b. **p->adu==q->adu** c. **p->adu==q** d. **p->adp==q->adp**

(Bacalaureat – Simulare 2006)

26. Dacă într-o listă liniară simplu înlățuită adresa de început a listei este **p**, iar adresa de sfârșit este **u**, atunci transformarea listei în listă circulară se realizează prin instrucțiunea:
 a. $p \rightarrow urm = u$ b. $p = u \rightarrow urm$ c. $u \rightarrow urm = p$ d. $u = p \rightarrow urm$
 (Bacalaureat – Sesiunea iunie-iulie 2004)
27. Se consideră o listă simplu înlățuită ale cărei noduri rețin în câmpul **urm** adresa nodului următor al listei sau NULL dacă nu există un nod următor. Pentru inserarea unui nod aflat la adresa **p** imediat după un nod al listei aflat la adresa **q**, se utilizează unele dintre următoarele atribuirii: 1) $p \rightarrow urm = q$; 2) $q \rightarrow urm = p$; 3) $p = q \rightarrow urm$; 4) $q = p \rightarrow urm$; 5) $p \rightarrow urm = q \rightarrow urm$; 6) $q \rightarrow urm = p \rightarrow urm$. Stabilită care dintre acestea se utilizează și în ce ordine:
 a. 3 6 b. 2 4 c. 5 2 d. 2 3
 (Bacalaureat – Simulare 2006)
28. Variabila **vf** memorează adresa elementului din vârful stivei. Fiecare element al stivei memorează într-un câmp **adr** adresa următorului element din stivă. Variabila **q** poate memora adresa oricărui element al stivei. Să se realizeze eliminarea elementului din vârful stivei:
 a. $q \rightarrow adr = vf; vf \rightarrow adr = q; \text{delete } q;$ b. $q = vf \rightarrow adr; vf = q \rightarrow adr; \text{delete } q;$
 c. $q = vf; vf = vf \rightarrow adr; \text{delete } vf;$ d. $q = vf; vf = q \rightarrow adr; \text{delete } q;$
 (Bacalaureat – Simulare 2003)
29. O listă liniară simplu înlățuită cu adresa de început memorată în variabila **p** este vidă dacă:
 a. $*p == \text{NULL} / ! *p$ b. $p == \text{NULL} / !p$
 c. $p != \text{NULL} / p$ d. $*p != \text{NULL} / *p$
 (Bacalaureat – Sesiunea specială 2003)
30. Într-o listă dublu înlățuită cu cel puțin 4 elemente, fiecare element reține în câmpul **adp** și **adu** adresa elementului precedent și respectiv următor din listă. Dacă **p** reprezintă adresa primului element din listă, iar **q** este de același tip cu **p**, atunci secvența alăturată realizează:
 a. interschimbarea primelor două componente b. eliminarea primului element
 c. eliminarea celui de-al doilea element d. eliminarea ultimului element
q = p → adu → adp;
p = q → adu;
p → adp = NULL;
delete q;
 (Bacalaureat – Sesiunea specială 2005)
31. Într-o listă dublu înlățuită cu cel puțin 4 elemente, fiecare element reține în câmpul **adp** și **adu** adresa elementului precedent și respectiv următor din listă. Dacă **p** reprezintă adresa primului element din listă, atunci $p \rightarrow adu \rightarrow adu \rightarrow adp$ este
 a. adresa primului element b. adresa celui de-al doilea element
 c. adresa celui de-al treilea element d. adresa celui de-al patrulea element
 (Bacalaureat – Simulare 2005)
32. Într-o listă circulară simplu înlățuită fiecare element reține în câmpul **next** adresa elementului următor. Știind că, pentru variabila **p** ce memorează adresa unui element oarecare din listă, este adevărată relația $p \rightarrow next = p$, atunci lista este formată din:
 a. zero componente b. o componentă c. 2 componente d. minim 3 componente
 (Bacalaureat – Sesiunea august-septembrie 2005)
33. Dacă **vf** indică ultimul nod al stivei, care dintre următoarele expresii trebuie să fie adevărate, pentru ca stiva să fie vidă?
 a. $vf == \text{NULL}$ b. $vf \rightarrow urm == 0$
 c. $vf \rightarrow urm == \text{NULL}$ d. $vf == 0$

34. Dacă **vf** indică ultimul nod al stivei, iar **q** un pointer către tipul nod, care dintre următoarele secvențe de instrucțiuni extrage nodul din vârful stivei?
- q=vf->urm; vf=q->urm; delete q;**
 - q=vf; vf=vf->urm; delete q;**
 - q=vf; vf=q->urm; delete q;**
 - q=vf; vf=q->urm; delete q;**
35. Dacă variabila **cap** memorează adresa primului nod din coadă și variabila **baza** memorează adresa ultimului nod din coadă, care dintre următoarele expresii trebuie să fie adevărate, pentru a avea o coadă vidă?
- baza==NULL**
 - cap->urm==baza**
 - cap==baza**
 - cap==NULL**

Miniproiecte:

Observație: Pentru realizarea următoarelor miniproiecte se va lucra în echipă. Profesorul va numi conducătorii de proiect, le va distribui proiectele și le va aloca un buget pentru realizarea lor (pentru simplificare, bugetul va fi folosit numai pentru plata membrilor echipei care vor realiza proiectul). Conducătorii de proiect vor negocia cu profesorul termenul de predare a aplicației, echipa cu care o vor realiza, și, dacă este cazul, bugetul care li s-a alocat inițial. Pe timpul realizării aplicațiilor, membrii echipei pot migra de la o echipă la alta, cu condiția să rămână încadrati într-o echipă, iar migrația să se facă numai cu acceptul conducătorilor echipei care migrează. Fiecare echipă va fi formată din:

- **Conducătorul proiectului** își va forma echipa și va distribui sarcinile pentru fiecare membru, negociind inițial suma repartizată din bugetul alocat pentru realizarea sarcinii. Va fixa termene de execuție pentru fiecare membru al echipei și va urmări modul în care sunt respectate aceste termene. În cazul în care unul dintre membrii echipei nu își realizează corect și la timp sarcinile, va redistribui o parte dintre sarcini între ceilalți membri ai echipei, renegociind suma din buget alocată fiecărui membru. La sfârșit, va da calitative fiecărui membru al echipei, în funcție de modul în care și-au respectat termenele, de modul în care au cooperat cu ceilalți membri ai echipei și de calitatea lucrărilor executate.
- **Analistul** va analiza cerințele informaționale ale aplicației, va determina funcțiile aplicației și va elabora modul de rezolvare (datele și structurile de date folosite, procesele în care este descompusă aplicația – care vor fi implementate cu subprograme – și meniul care asigură interfața cu utilizatorul). În realizarea acestor sarcini va fi ajutat și îndrumat de conducătorul proiectului.
- **Grupul de programatori** (numărul lor trebuie să fie stabilit în funcție de dimensiunea proiectului) va implementa în limbajul de programare soluția găsită de analistul echipei. Conducătorul proiectului le va repartiza subprogramele pe care le vor realiza și specificațiile fiecărui subprogram: datele de intrare, datele de ieșire și funcția subprogramului.
- **Testorul** va testa aplicația. El va trebui să aleagă seturi de date de intrare astfel încât să găsească erorile de logică și de execuție ale aplicației.
- **Documentaristul** va întocmi documentațiile aplicației: documentația pentru beneficiari și documentația pentru proiectantul aplicației.

La terminarea proiectului membrii echipei vor primi note pentru evaluarea activității lor. Sistemul de evaluare trebuie să țină cont de veniturile realizate pentru munca depusă, de calificativul obținut de la conducătorul de proiect și de calitatea muncii evaluată de profesor.

- Pentru biblioteca școlii sunt aduse cărți de la mai multe edituri. Cărțile trebuie organizate în ordinea alfabetică a autorilor, și pe fiecare autor în ordinea alfabetică a titlurilor. Este posibil ca pentru același autor și același titlu să se primească mai multe exemplare. Se va folosi căte o stivă pentru fiecare autor, în care se va simula teancul de titluri primite, pentru fiecare titlu memorându-se și numărul de exemplare. Numele autorilor și adresa

vârfului stivei de cărți asociate se vor memora într-o listă ale cărei elemente conțin în informația utilă două câmpuri: un câmp de tip sir de caractere pentru numele autorului și un câmp de tip pointer către tipul nod al stivei pentru vârful stivei. Scrieți o aplicație care să asigure următoarele operații prin intermediul unui meniu:

- Distribuirea pe autori și titluri a unui teanc de cărți sosit de la o editură.
- Afișarea titlurilor și a numărului de exemplare ale unui autor al căruia nume se citește de la tastatură.
- Afișarea în ordine alfabetică a autorilor și a numărului de titluri și de exemplare pentru fiecare autor.
- Numele autorilor cu cele mai multe, respectiv cu cele mai puține titluri.
- Numele autorilor cu cele mai multe, respectiv cu cele mai puține exemplare.

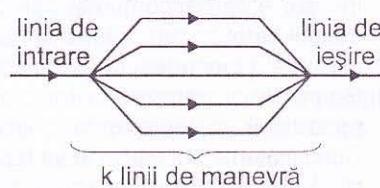
2. La un concurs participă mai mulți candidați identificați după nume și prenume. Fiecare candidat primește la înscriere un număr de identificare. Concursul constă în trei probe, notele putând lua valori de la 1 la 10. Rezultatul concursului se stabilește pe baza mediilor aritmetice a notelor primite. Trebuie prevăzute două variante de admitere a candidaților: sunt admisi toți candidații care au o medie mai mare decât m și sunt admisi, în ordinea mediilor, primii k candidați (valorile pentru m și k se citesc de la tastatură). Scrieți o aplicație care să asigure următoarele operații prin intermediul unui meniu:

- Înscrierea unui nou candidat la concurs.
- Retragerea unui candidat din concurs.
- Completarea notelor și calcularea mediei pentru fiecare candidat.
- Modificarea informațiilor despre un candidat.
- Afișarea candidaților admisi în fiecare dintre cele două variante în ordinea descrescătoare a mediilor.
- Afișarea candidaților admisi în fiecare dintre cele două variante în ordinea alfabetică a numelui și prenumelui.

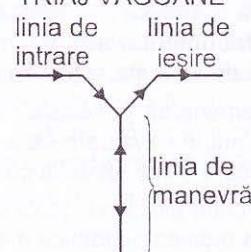
3. Într-un depou există o linie pe care se găsesc mai multe locomotive, aranjate în ordinea în care au intrat în depou, și o linie pe care se găsesc mai multe vagoane, aranjate în ordinea în care au intrat în depou. Fiecare locomotivă și fiecare vagon are un număr de identificare. În plus, pentru fiecare vagon este precizată și clasa (clasa 1 și clasa 2). Pentru linia care conține locomotive există un triaj cu k linii de manevră. Valoarea pentru k se citește de la tastatură. În triajul vagoanelor există o linie de intrare și o linie de ieșire pe care deplasarea se poate face numai în sensul săgeților și o linie de manevră pe care deplasarea se poate face în ambele sensuri. Scrieți o aplicație care să asigure următoarele operații prin intermediul unui meniu:

- Intrarea unei locomotive în depou.
- Afișarea locomotivelor din depou.
- Intrarea unui vagon în depou.
- Afișarea vagoanelor din depou, precizându-se căte vagoane sunt de clasa 1 și căte sunt de clasa 2.
- Formarea unei garnituri. Garnitura este formată dintr-o locomotivă cu numărul de identificare p și n vagoane, dintre care m vagoane sunt de clasa 1 (valorile pentru p , n și m se citesc de la tastatură).

TRIAJ LOCOMOTIVE



TRIAJ VAGOANE



2.7. Graful

2.7.1. Definiția matematică a grafului

Se numește **graf (G)** o pereche ordonată de mulțimi (X, U) , unde X este o mulțime finită și nevidă, iar U o mulțime de perechi formate cu elemente **distincte** din mulțimea X (familie de submulțimi cu două elemente din mulțimea X).

Terminologie:

→ Elementele mulțimii X se numesc **vârfuri** sau **noduri**. Mulțimea X se mai numește și **mulțimea vârfurilor** sau **mulțimea nodurilor** grafului G . Ea este de forma:

$$X = \{x_1, x_2, x_3, \dots, x_i, \dots, x_n\}$$

unde x_i reprezintă nodul i al grafului G care are n noduri.

→ **Ordinul grafului** reprezintă numărul de noduri ale grafului, n :

$$\text{ordinul grafului} = \text{card}(X) = n$$

→ Elementele mulțimii U sunt perechi de noduri, adică submulțimi cu două elemente din mulțimea X și se notează cu u_k . Elementul u_k este definit de perechea de forma $\{x_i, x_j\}$, unde $x_i, x_j \in X$ și $x_i \neq x_j$ (elemente distincte din mulțimea X). Elementul u_k leagă nodurile x_i și x_j și se notează astfel: $[x_i, x_j]$. Mulțimea U este de forma:

$$U = \{u_1, u_2, u_3, \dots, u_k, \dots, u_m\}$$

Clasificarea grafurilor:

Criteriul de clasificare folosit este **proprietatea de simetrie** a mulțimii U .

Mulțimea U are **proprietatea de simetrie** dacă și numai dacă, pentru orice pereche de noduri (x_i, x_j) , dacă $\{x_i, x_j\} \in U$, atunci și $\{x_j, x_i\} \in U$

În funcție de proprietatea de simetrie, grafurile se clasifică în:

→ **Grafuri neorientate**. Un graf $G=(X,U)$ este un graf neorientat dacă mulțimea U are proprietatea de simetrie. Mulțimea U este formată din **perechi neordonate** $\{x_j, x_i\}$.

→ **Grafuri orientate**. Un graf $G=(X,U)$ este un graf orientat dacă mulțimea U nu are proprietatea de simetrie. Mulțimea U este formată din **perechi ordonate** $\{x_j, x_i\}$.

Pentru a identifica tipul de graf pe care îl veți folosi pentru a reprezenta datele, **definiți relația dintre nodurile grafului** și verificați dacă relația are proprietatea de simetrie, astfel:

→ Dacă nodul x în relație cu nodul y implică și că nodul y este în relație cu nodul x , atunci graful este neorientat.

→ Dacă nodul x în relație cu nodul y nu implică și că nodul y este în relație cu nodul x , atunci graful este orientat.

Studiu de caz

Scop: identificarea tipului de graf pe care îl folosiți pentru a rezolva problema.

Enunțul problemei 1. Pe harta unui județ există mai multe localități care sunt legate prin șosele pe care se circulă în ambele sensuri. Să se identifice traseele pe care se poate ajunge de la localitatea A la localitatea B.

Nodurile grafului sunt localitățile. Relația care se stabilește între nodurile grafului este: nodul x este în relație cu nodul y , dacă există o șosea care leagă direct localitatea asociată nodului x cu localitatea asociată nodului y . Relația are proprietatea de simetrie, deoarece șoseaua care leagă direct localitatea asociată nodului x cu localitatea asociată nodului y leagă direct și localitatea asociată nodului y cu localitatea asociată nodului x . Pentru reprezentarea căilor de comunicație dintre localități se va folosi un **graf neorientat**.

Enunțul problemei 2. Pe harta unui cartier există mai multe intersecții care sunt legate de străzi. Pe unele străzi se poate circula în ambele sensuri, pe alte străzi numai într-un anumit sens. Să se identifice traseele prin care se poate ajunge de la intersecția A la intersecția B.

Nodurile grafului sunt intersecțiile. Relația care se stabilește între nodurile grafului este: nodul x este în relație cu nodul y , dacă există trafic care leagă direct intersecția asociată nodului x cu intersecția asociată nodului y (se poate circula de la nodul x la nodul y). Relația nu are proprietatea de simetrie deoarece, dacă există o stradă care leagă direct intersecția asociată nodului x cu intersecția asociată nodului y și pe această stradă există trafic de la nodul x la nodul y , nu este obligatoriu ca pe acea stradă să existe trafic și de la nodul y la nodul x . Pentru reprezentarea traficului auto dintre intersecții se va folosi un **graf orientat**.

Enunțul problemei 3. La nivelul unui grup de persoane se face un studiu social. Între persoane se stabilesc relații de prietenie, dar și relații de simpatie. Să se descrie cu ajutorul grafului relațiile dintre persoane.

Nodurile grafului sunt membrii grupului de persoane. Între persoane se pot stabili relațiile:

- Relația de prietenie este o relație definită astfel: persoana x este în relație cu persoana y , dacă este prietenă cu ea. Relația este simetrică deoarece, dacă persoana x este prietenă cu persoana y , atunci și persoana y este prietenă cu persoana x (relația de prietenie presupune reciprocitate). Pentru reprezentarea relațiilor de prietenie dintre membrii grupului se va folosi un **graf neorientat**.
- Relația de simpatie este o relație definită astfel: persoana x este în relație cu persoana y , dacă o simpatizează. Relația nu este simetrică deoarece, dacă persoana x simpatizează persoana y , nu este obligatoriu ca persoana y să simpatizeze persoana x (relația de simpatie nu presupune reciprocitate). Pentru reprezentarea relațiilor de simpatie dintre membrii grupului se va folosi un **graf orientat**.



1. Prin ce tip de graf va fi reprezentat un grup de persoane între care s-au stabilit relații de vecinătate?
2. Prin ce tip de graf va fi reprezentat un grup de persoane între care s-au stabilit relații de cunoștință?



2.7.2. Graful neorientat

2.7.2.1. Terminologie

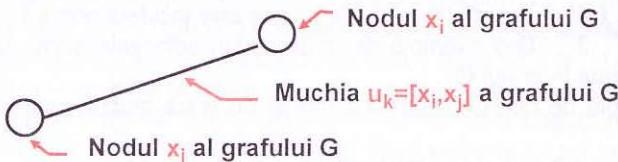
→ Elementele mulțimii U (perechile de noduri) se numesc **muchii**. Mulțimea U se mai numește și **multimea muchiilor** grafului G . O muchie, fiind un element din mulțimea U , este determinată de o submulțime cu două elemente din mulțimea X : muchia k a grafului (U_k), care unește nodurile x_i și x_j , este determinată de submulțimea $\{x_i, x_j\}$ și se notează cu $[x_i, x_j]$. $[x_i, x_j]$ și $[x_j, x_i]$ reprezintă aceeași muchie a grafului. Graful G are m muchii:

$$\text{numărul de muchii} = \text{card}(U) = m$$

- Numim **noduri adiacente** orice pereche de noduri care formează o muchie – $\{x_i, x_j\} \in U$. Fiecare dintre cele două noduri (x_i și x_j) este **nod incident** cu muchia $u_k = [x_i, x_j]$.
- **Nodurile vecine** unui nod x_i sunt toate nodurile x_j care sunt adiacente cu el.
- Se numește **nod extrem** al unei muchii oricare dintre cele două noduri care se găsesc la capătul muchiei. Nodurile x_i și x_j sunt **extremitățile** muchiei $[x_i, x_j]$.
- Se numesc **muchii incidente** două muchii u_i și u_j care au o extremitate comună – nodul x_k .

Un graf neorientat G este definit de o pereche de multimi: multimea nodurilor sale – X și multimea muchiilor sale – U . El poate fi considerat ca o multime de noduri din care unele pot fi unite două căte două printr-o muchie.

Graful se reprezintă în plan prin intermediul unor elemente geometrice: nodurile se reprezintă prin cercuri, iar muchiile prin linii drepte care unesc anumite cercuri.



Elementele multimii X (nodurile) se identifică cu ajutorul unor **etichete**, care pot fi numere sau litere. Pentru simplificare, vom folosi ca etichete un sir de numere consecutive, începând cu numărul 1. De exemplu, pentru un graf cu n noduri, vom folosi etichetele: 1, 2, 3, ..., $n-1$, n . O muchie se va nota cu $[i,j]$, unde i și j sunt etichetele nodurilor incidente cu muchia. De exemplu, muchia $[2,3]$ este muchia care unește nodurile cu etichetele 2 și 3.

Exemplul 1:

În graful $G_1=(X_1, U_1)$ din figura 1:

- **Ordinul grafului** este 8.
- **Graful** are 8 noduri ($n=8$) și **multimea nodurilor** este $X_1=\{1,2,3,4,5,6,7,8\}$.
- **Graful** are 9 muchii ($m=9$) și **multimea muchiilor** este $U_1=\{[1,2], [1,3], [1,4], [2,3], [2,5], [3,4], [3,5], [6,7], [6,8]\}$.
- Nodul 1 este **nod adiacent** cu nodurile 2, 3 și 4, iar nodul 6 este adiacent cu nodurile 7 și 8. Nodurile 3 și 4 sunt adiacente deoarece perechea de noduri $[3,4] \in U_1$. Nodurile 5 și 6 nu sunt adiacente deoarece perechea de noduri $[5,6] \notin U_1$.
- Nodul 5 este **nod incident** cu muchiile $[2,5]$ și $[3,5]$, dar nu este incident cu muchia $[1,2]$.
- Nodul 3 este **nod extrem** muchiilor $[1,3]$, $[2,3]$, $[3,4]$ și $[3,5]$.
- Muchiile $[1,2]$ și $[2,3]$ sunt **muchii incidente** deoarece au un nod comun (nodul 2). Muchiile $[1,4]$ și $[2,3]$ nu sunt muchii incidente deoarece nu au un nod comun.

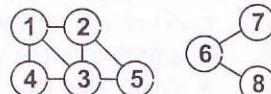


Fig. 1

Teorema 1

Dacă graful neorientat G are n noduri (x_1, x_2, \dots, x_n), atunci **numărul total de grafuri neorientate care se pot forma cu aceste noduri este** $g = 2^{C_n^2}$

$$g = 2^{C_n^2}$$

Demonstrație. Notăm cu X multimea nodurilor grafului, cu U multimea muchiilor, cu A multimea tuturor submulțimilor de două elemente din X și cu B multimea $\{0,1\}$. Multimea A are următoarele elemente (submulțimi):

[1,2], [1,3], [1,4], ..., [1,n]	n-1 submulțimi
[2,3], [2,4], ..., [2,n]	n-2 submulțimi
[n-1,n]	1 submulțime

Numărul total de submulțimi este: $(n-1) + (n-2) + \dots + 1 = \frac{n \times (n-1)}{2} = C_n^2$

Notăm cu $a = \text{card}(A)$ și cu $b = \text{card}(B)$. Fiecărui graf îl putem asocia o funcție $f: A \rightarrow B$ definită astfel. Invers, unei funcții $f: A \rightarrow B$ îl putem atașa un graf, astfel: $f(\{x,y\}) = 1$ dacă și numai dacă $[x,y] \in U$. Rezultă că numărul total de grafuri care se pot forma cu n noduri este egal cu

numărul de funcții f . Dar, numărul de funcții $f: A \rightarrow B$ este egal cu b^a , unde $b=2$ și $a=C_n^2$

Tema

- În graful G_1 – cu ce noduri este adiacent nodul 1?
- În graful G_1 – cu ce muchii este incident nodul 1?
- Dați exemple de două noduri adiacente și de două noduri care nu sunt adiacente în graful G_1 .
- Dați exemplu de două muchii incidente și de două muchii care nu sunt incidente în graful G_1 .
- Desenați graful $G_2 = (X_2, U_2)$ definit astfel:
 $X_2 = \{1, 2, 3, 4, 5, 6, 7, 8\}$
 $U_2 = \{\{1, 2\}, \{1, 3\}, \{1, 5\}, \{2, 3\}, \{2, 5\}, \{3, 4\}, \{4, 5\}, \{4, 6\}, \{4, 7\}\}$.
- Desenați graful traseelor rutiere care fac legătura între localitățile Brașov, București, Buzău, Ploiești și Constanța. Dacă există mai multe trasee rutiere între două localități (de exemplu, București și Brașov), adăugați la graf noduri pentru localitățile care identifică unic aceste trasee (de exemplu, Vălenii de Munte, Târgoviște și Pitești).
- Desenați graful județelor din România (între două județe există o muchie dacă cele două județe sunt învecinate).
- Câte grafuri se pot construi cu 3 muchii? Desenați toate grafurile care se pot construi cu 3 muchii.
- Pentru graful G_3 din figura 2, precizați ordinul, numărul de noduri, numărul de muchii și mulțimile X_3 și U_3 .
- Structura unei molecule de substanță chimică poate fi reprezentată printr-un graf neorientat, în care nodurile sunt atomii și grupările din care este compusă molecula, iar muchiile sunt legăturile dintre ele.
În figura 3 este prezentat graful moleculei de apă H_2O . Reprezentați grafurile moleculelor de H_2SO_4 , NH_3 , CH_4 și C_2H_4 .

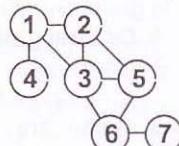


Fig. 2

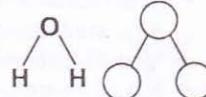


Fig. 3

2.4.2.2. Gradul unui nod al grafului neorientat

Nodul unui graf este caracterizat prin grad.

Gradul unui nod x_k al grafului G este egal cu numărul muchiilor incidente cu nodul și se notează cu $d(x_k)$.

Terminologie:

- Se numește **nod terminal** un nod care are gradul egal cu 1 – $d(x_k) = 1$ (este incident cu o singură muchie).
- Se numește **nod izolat** un nod care are gradul egal cu 0 – $d(x_k) = 0$ (nu este adiacent cu nici un alt nod al grafului, adică nu se găsește în extremitatea nici unei muchii).

Exemplul 1:

Graful $G_4 = (X_4, U_4)$ din figura 4 este definit astfel:

$$X_4 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

$$U_4 = \{[1, 2], [1, 4], [2, 3], [2, 5], [3, 4], [3, 5], [5, 6], [5, 7], [5, 8], [7, 9]\}.$$

În graful G_4 :

- **Gradul** nodului 5 este 5, deoarece are 5 muchii incidente: [2, 5], [3, 5], [5, 6], [5, 7] și [5, 8].
- Nodul 9 este **nod terminal**, deoarece are gradul 1 (o singură muchie incidentă: [7, 9]).
- Nodul 10 este **nod izolat**, deoarece are gradul 0 (nicio muchie incidentă).

Exemplul 2:

Fie graful $G_5 = (X_5, U_5)$, unde $X_5 = \{1, 2, 3, 4, 5, 6, 7, 8\}$ și $U_5 = \{[1, 2], [1, 5], [2, 3], [2, 5], [3, 4], [3, 5], [4, 5], [4, 6], [4, 7]\}$. Din lista muchiilor unui graf neorientat, puteți preciza următoarele informații:

- Determinați **gradul** unui nod – numărând de câte ori apare eticheta nodului în lista de muchii. Nodul 5 are gradul 3 (în mulțimea muchiilor, eticheta 5 apare de 3 ori: [1, 5], [2, 5], [3, 5]).
- Determinați dacă un nod este terminal – verificând dacă eticheta lui apare o singură dată. Nodul 7 este **nod terminal** (eticheta lui apare numai într-o singură muchie: [4, 7]).
- Determinați dacă un nod este izolat – verificând dacă eticheta lui nu apare în lista de muchii. Nodul 8 este **nod izolat** (eticheta lui nu apare în lista muchiilor).
- Determinați **numărul de noduri izolate** (n_1) astfel: numărați etichetele distincte care apar în lista muchiilor (n_2) și $n_1 = n - n_2$. În graful G_5 , în lista de muchii există 7 etichete distincte. Numărul de noduri izolate este 1 (8-7).

Observație: Într-un graf cu n noduri, oricare ar fi nodul x_k , gradul său este mai mare sau egal cu 0 și mai mic sau egal cu $n-1$ ($0 \leq d(x_k) \leq n-1$).



1. În graful G_4 : precizați gradul nodului 3 și identificați nodul cu gradul cel mai mare, nodurile izolate și nodurile terminale.
2. În graful G_3 : identificați nodurile care au gradul 2, precizați câte noduri au gradul impar și care este nodul cu gradul cel mai mare.
3. În graful G_5 : precizați gradul nodului 3, identificați nodurile izolate și nodurile terminale, precizați câte noduri au gradul 2 și câte noduri au gradul impar.

Teorema 2

Dacă graful G are m muchii (u_1, u_2, \dots, u_m) și n noduri (x_1, x_2, \dots, x_n), atunci între gradul nodurilor și numărul de muchii există următoarea relație: **suma gradelor tuturor nodurilor grafului este egală cu dublul numărului de muchii**:

$$\sum_{i=1}^n d(x_i) = 2m$$

Demonstrație. Fiecare muchie $u_k = [x_i, x_j]$ corespunde unei unități din gradul nodului x_i și unei unități din gradul nodului x_j . Rezultă că fiecare muchie contribuie cu 2 unități la suma gradelor.

Exemplu. În graful G_4 : $d(1) + d(2) + d(3) + d(4) + d(5) + d(6) + d(7) + d(8) + d(9) + d(10) + d(11) = 2+2+3+2+4+1+2+1+1+0+0 = 18 = 2 \times 9 = 2 \times m$

Propoziția 1. Pentru orice graf G , **numărul nodurilor de grad impar este par**.

Demonstrație. Suma gradelor nodurilor fiind un număr par, această sumă trebuie să conțină un număr par de termeni care sunt numere impare.

Exemplu. În graful G_3 există 4 noduri cu grad impar (3, 6, 8 și 9).

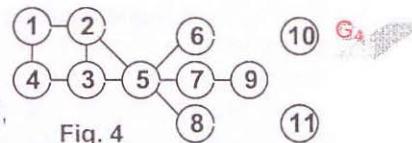


Fig. 4

G_4

G_5

Propoziția 2. Numărul minim de muchii, m_{\min} , pe care trebuie să le aibă un graf neorientat, cu n noduri, ca să nu existe noduri izolate, este:

$$m_{\min} = \left[\frac{n+1}{2} \right]$$

Demonstrație. Pentru ca un nod x_i să nu fie izolat, trebuie ca $d(x_i) \geq 1$. Pentru ca toate nodurile să nu fie izolate, trebuie ca suma gradelor să fie mai mare sau egală cu n . Dar, suma gradelor este dublul numărului de muchii – m . Înseamnă că, pentru n par – $m_{\min} = n/2$, iar pentru n impar – $m_{\min} = (n+1)/2$.

Teorema 3

Dacă graful G are n noduri ($n \geq 2$), atunci **cel puțin două noduri au același grad**.

Demonstrație – prin reducere la absurd. Presupunem că nu este adevărat. Cum, oricare ar fi nodul x_k , $0 \leq d(x_k) \leq n-1$, înseamnă că singurul sir de n numere, diferite între ele două căte două, care pot reprezenta gradele unghiurilor este $0, 1, 2, \dots, n-1$. Deoarece un nod este izolat, cel mai mare grad al unui nod nu poate fi decât $n-2$ (nodul nu se poate lega de el însuși și de nodul izolat). Rezultă că sirul de numere definit mai sus (singurul sir care se poate defini) nu poate reprezenta sirul gradelor în graf.

- 
1. În graful G_1 – verificați că este îndeplinită relația dintre gradul nodurilor și numărul de muchii ale grafului. Identificați nodurile cu grad impar și verificați că numărul lor este par.
 2. Dacă un graf are 8 noduri, care este numărul minim de muchii pe care trebuie să le aibă, ca să nu fie noduri izolate. Desenați un graf care, având numărul minim de muchii stabilit, nu conține noduri izolate. Dacă un graf are 9 noduri, care este numărul minim de muchii pe care trebuie să le aibă, ca să nu fie noduri izolate. Desenați un graf care având numărul minim de muchii stabilit nu conține noduri izolate.



2.7.2.3. Sirul grafic

Se numește **sir grafic** un sir s de n numere întregi pozitive (d_1, d_2, \dots, d_n) care pot reprezenta gradele unui graf neorientat, cu n noduri.

Propoziția 3

Condițiile necesare ca un sir de n numere întregi pozitive (d_1, d_2, \dots, d_n) să fie un sir grafic sunt:

- (1) $d_i \leq n-1$, pentru orice $i=1,n$;
- (2) suma $d_1 + d_2 + \dots + d_n$ trebuie să fie un număr par.

Demonstrație. Necesitatea condiției (1) rezultă din faptul că gradul maxim al unui nod dintr-un graf cu n noduri poate fi $n-1$. Necesitatea condiției (2) rezultă din Teorema 2 – suma gradelor fiind egală cu dublul numărului de muchii, este un număr par.

Acstea condiții nu sunt întotdeauna și suficiente. Pentru a verifica dacă sirul de numere este sir grafic, se face analiza sirului de numere.

Exemple:

(a) $s=(1,1,2,3,3,4,5,5,7,8)$

Acest sir nu îndeplinește una dintre condițiile necesare – (2) – suma numerelor este 39.

(b) $s=(1,1,1,2,2,3,5,6,8,9)$

Acest sir îndeplinește condițiile necesare (suma numerelor este 38 și fiecare număr este mai mic sau egal cu 9: $9=10-1$). Aceste condiții nu sunt însă suficiente. Din analiza sirului rezultă că nodul 10, având gradul 9, este legat de toate celelalte 9 noduri. Nodurile 1, 2 și 3 sunt noduri terminale. Ele nu pot fi legate decât de nodul 10. Rezultă că gradul maxim

pe care îl poate avea oricare dintre celelalte șase noduri este 6 (ele nu se pot lega de ele însăși și de nodurile 1, 2 și 3). Dar nodul 9 are gradul 8, ceea ce este imposibil.

(c) $s=(1,1,1,2,3,4,5,6,6,9)$

Și acest sir îndeplinește condițiile necesare (suma numerelor este 38 și fiecare număr este mai mic sau egal cu 9). În plus, față de sirul (b), având aceleași grade pentru nodurile 1, 2, 3 și 10, îndeplinește și condiția ca celelalte noduri să aibă gradul mai mic sau egal cu gradul maxim posibil (6). Dar, există două noduri cu gradul 6. Ele trebuie să se lege amândouă de nodul 4, la care este legat și nodul 10. Nodul 4 trebuie să aibă cel puțin gradul 3. Dar nodul 4 are gradul 2, ceea ce este imposibil.

(d) $s=(1,1,1,3,3,4,4,5,5,9)$

Acest sir îndeplinește condițiile necesare (suma numerelor este 38 și fiecare număr este mai mic sau egal cu 9) și este un sir grafic căruia i se poate asocia graful $G_6=(X_6, U_6)$, cu 18 muchii, definit astfel:

$$X_6=\{1,2,3,4,5,6,7,8,9,10\}$$

$$U_6=\{[1,10], [2,10], [3,10], [4,8], [4,9], [4,10], [5,8], [5,9], [5,10], [6,7], [6,8], [6,9], [6,10], [7,8], [7,9], [7,10], [8,10], [9,10]\}.$$

Tema



Precizați dacă sirurile $s_1=(1,1,2,2,4)$ și $s_2=(0,1,1,1,4)$ pot fi siruri grafice. Pentru sirul care este sir grafic, găsiți un graf care i se poate asocia.

2.7.3. Graful orientat

Spre deosebire de graful neorientat, în graful orientat perechile de noduri sunt ordonate. Graful orientat se mai numește și **digraf**.

2.7.3.1. Terminologie

→ Elementele multimii U (perechile de noduri) se numesc **arce**. Multimea U se mai numește și **multimea arcelor** grafului G . Un arc, fiind un element din multimea U , este determinat de o submultime ordonată, cu două elemente, din multimea X : arcul k al grafului (u_k), ce unește nodurile x_i și x_j , este determinat de submultimea $\{x_i, x_j\}$ și se notează cu $[x_i, x_j]$. $[x_i, x_j]$ și $[x_j, x_i]$ nu reprezintă același arc al grafului. Graful G are m arce:

$$\text{numărul de arce} = \text{card}(U) = m$$

→ Se numesc **noduri adiacente** în graful G oricare din perechile de noduri care formează un arc – $(x_i, x_j) \in U$ sau $(x_j, x_i) \in U$. Fiecare dintre cele două noduri (x_i și x_j) este **nod incident** cu arcul $u_k = [x_i, x_j]$ sau cu arcul $u_k = [x_j, x_i]$.

→ Nodurile x_i și x_j sunt extremitățile arcului $[x_i, x_j]$. Nodul x_i este **extremitatea inițială** a arcului, iar nodul x_j este **extremitatea finală** a arcului.

→ Se numesc **arce incidente** două arce u_i și u_j care au o extremitate comună – nodul x_k .

→ Se numește **succesor** al nodului x_i orice nod la care ajunge un arc careiese din nodul x_i . **Multimea succesorilor** nodului x_i este formată din multimea nodurilor la care ajung arcele care ies din nodul x_i . Se notează cu $S(x_i)$ și se definește ca multimea:

$$S(x) = \{x_j \in X \mid (x_i, x_j) \in U\}.$$

→ Se numește **predecesor** al nodului x_i orice nod de la care intră un arc în nodul x_i . **Multimea predecesorilor** nodului x_i este formată din multimea nodurilor de la care ajung arcele care intră în nodul x_i . Se notează cu $P(x_i)$ și se definește ca multimea:

$$P(x) = \{x_j \in X \mid (x_j, x_i) \in U\}.$$

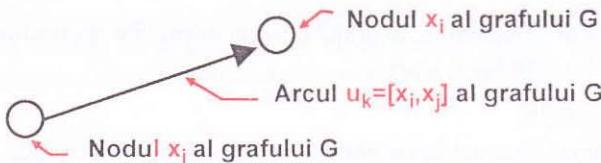
- **Mulțimea arcelor care ieș** din nodul x_i se notează cu $U^+(x_i)$ și se definește ca mulțimea $U^+(x_i) = \{u=(x_i, x_j) \mid u \in U\}$.
- **Mulțimea arcelor care intră** în nodul x_i se notează cu $U^-(x_i)$ și se definește ca mulțimea $U^-(x_i) = \{u=(x_j, x_i) \mid u \in U\}$.
- **Nodul sursă** al grafului este nodul care are mulțimea succesorilor formată din toate celelalte noduri, mai puțin el, iar mulțimea predecesorilor săi este mulțimea vidă.
- **Nodul destinație** al grafului este nodul care are mulțimea predecesorilor formată din toate celelalte noduri, mai puțin el, iar mulțimea succesorilor săi este mulțimea vidă.

Observații

1. $\text{card}(S(x)) = \text{card}(U^+(x))$ și $\text{card}(P(x)) = \text{card}(U^-(x))$.
2. Pentru nodul sursă al grafului $\text{card}(S(x)) = \text{card}(X)-1$ și $\text{card}(P(x))=0$.
3. Pentru nodul destinație al grafului $\text{card}(P(x)) = \text{card}(X)-1$ și $\text{card}(S(x))=0$.
4. Dacă un graf are un nod sursă, atunci nu poate avea un nod destinație, și invers.

Un graf orientat G este definit de o pereche de mulțimi: mulțimea nodurilor sale – X și mulțimea arcelor sale – U . El poate fi considerat ca o mulțime de noduri din care unele pot fi unite două câte două, prin unul sau două arce.

Graful orientat se **reprezintă în plan** prin intermediul unor elemente geometrice: nodurile se reprezintă prin cercuri, iar arcele prin linii drepte care unesc anumite cercuri și care au o săgeată la capătul care corespunde extremității finale a arcului.



Exemplu:

G7

În graful $G_7 = (X_7, U_7)$ – din figura 5:

- Ordinul **grafului** este 5.
- Graful are 5 noduri ($n=5$) și **mulțimea nodurilor** este $X_7 = \{1, 2, 3, 4, 5\}$.
- Graful are 7 arce ($m=7$) și **mulțimea arcelor** este $U_7 = \{[1, 2], [1, 4], [2, 3], [4, 1], [4, 3], [5, 2], [5, 3]\}$.
- Nodul 1 este **nod adiacent** cu nodurile 2 și 4, iar nodul 3 este adiacent cu nodurile 2, 4 și 5. Nodurile 3 și 4 sunt adiacente deoarece perechea de noduri $[4, 3] \in U_7$. Nodurile 5 și 4 nu sunt adiacente, deoarece nici una dintre perechile de noduri $[4, 5]$ și $[5, 4] \notin U_7$.
- Nodul 4 este **nod incident** cu arcele $[1, 4]$, $[4, 1]$ și $[4, 3]$, dar nu este incident cu arcul $[1, 2]$.
- Nodul 2 este **extremitatea inițială** a arcului $[2, 3]$ și **extremitatea finală** a arcului $[1, 2]$ și $[5, 2]$.
- Arcele $[1, 2]$ și $[5, 2]$ sunt **arce incidente** deoarece au un nod comun (nodul 2). Arcele $[1, 4]$ și $[2, 3]$ nu sunt arce incidente, deoarece nu au un nod comun.
- **Mulțimea succesorilor** nodului 1 este formată din nodurile 2 și 4. Nodul 2 este **nod succesor** al nodului 1, dar și al nodului 5. Nodul 1 este nod succesor al nodului 4, dar și nodul 4 este nod succesor al nodului 1. Nodul 3 nu are succesi.
- **Mulțimea predecesorilor** nodului 3 este formată din nodurile 2, 4 și 5. Nodul 2 este **nod predecesor** al nodului 3. Nodul 1 este nod predecesor al nodului 4, dar și nodul 4 este nod predecesor al nodului 1. Nodul 5 nu are predecesori.

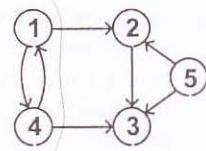


Fig. 5

Teorema 4

Dacă graful orientat G are n noduri (x_1, x_2, \dots, x_n) , atunci numărul total de grafuri orientate care se pot forma cu aceste noduri este g :

$$g = 4^{\frac{n \times (n-1)}{2}}$$

Demonstrație. Se demonstrează la fel ca Teorema 1, cu deosebirea că mulțimea B este $\{0, 1, 2, 3\}$ – $\text{card}(B)=4$, iar funcția f este definită astfel:

$$f(\{x, y\}) = \begin{cases} 3, & \text{dacă } [x, y] \in U \text{ și } [y, x] \in U \\ 2, & \text{dacă } [x, y] \in U \\ 1, & \text{dacă } [y, x] \in U \\ 0, & \text{dacă } [x, y] \notin U \text{ și } [y, x] \notin U \end{cases}$$

Temă

1. În graful G_7 – cu ce noduri este adiacent nodul 2?
2. În graful G_7 – cu ce arce este incident nodul 2?
3. Dați exemplu de două noduri adiacente în graful G_7 .
4. Dați exemplu de două noduri care nu sunt adiacente în graful G_7 .
5. Dați exemplu de două arce incidente în graful G_7 .
6. Dați exemplu de două arce care nu sunt incidente în graful G_7 .
7. În graful G_7 precizați ce fel de extremitate este nodul 4 pentru fiecare arc cu care este incident. Precizați mulțimea succesorilor și mulțimea predecesorilor nodului 4.
8. Desenați graful $G_8=(X_8, U_8)$, definit astfel.
 $X_8=\{1, 2, 3, 4, 5, 6, 7\}$
 $U_8=\{[1, 2], [1, 5], [2, 1], [2, 4], [3, 4], [4, 3], [5, 3], [6, 5], [6, 7], [7, 5]\}$.
9. Câte grafuri orientate se pot construi cu 3 arce? Desenați 10 dintre aceste grafuri.
10. Pentru graful G_9 din figura 6, precizați ordinul, numărul de noduri, numărul de arce și mulțimile X_9 și U_9 .

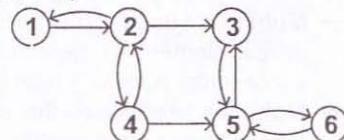
G₈

Fig. 6

2.7.3.2. Gradele unui nod al grafului orientat

Nodul unui graf orientat este caracterizat prin **gradul intern** și **gradul extern**.

Gradul intern al unui nod x_i al grafului G este egal cu numărul arcelor care intră în nodul x_i (arce de forma $[x_i, x_j]$) și se notează cu $d^-(x)$.

Gradul extern al unui nod x_i al grafului G este egal cu numărul arcelor care ies din nodul x_i (arce de forma $[x_i, x_j]$) și se notează cu $d^+(x)$.

Terminologie:

- Se numește **nod terminal** un nod care are suma gradelor egală cu 1 (gradul intern sau gradul extern egal cu 1 și gradul intern, respectiv gradul extern, egal cu 0 – $d^+(x_k) = 1$ și $d^-(x_k) = 0$ sau $d^-(x_k) = 1$ și $d^+(x_k) = 0$). Nodul terminal este incident cu un singur arc.
- Se numește **nod izolat** un nod care are suma gradelor egală cu 0 (gradul intern și gradul extern egale cu 0 – $d^+(x_k) = d^-(x_k) = 0$). Nodul izolat nu este adiacent cu nici un alt nod al grafului, adică nu se găsește la extremitatea niciunui arc.

Observații:

1. $d^+(x)=\text{card}(S(x))$ și $d^-(x)=\text{card}(P(x))$.
2. Dacă graful are n noduri, pentru un nod sursă al grafului $d^+(x)=n-1$ și $d^-(x)=0$, iar pentru un nod destinație al grafului $d^-(x)=n-1$ și $d^+(x)=0$.

Exemplul 1:

G₁₀ Graful $G_{10} = (X_{10}, U_{10})$ din figura 7 este definit astfel:

$$X_{10} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

$$U_{10} = \{[1, 2], [1, 4], [2, 1], [2, 3], [2, 5], [2, 6], [2, 7], [4, 1], [7, 2], [8, 9], [9, 8]\}.$$

În graful G_{10} :

- **Gradul intern** al nodului 2 este 2, deoarece are 2 arce care intră: [1, 2] și [7, 2]. **Gradul extern** al nodului 2 este 4, deoarece are 4 arce care iau: [2, 1], [2, 3], [2, 5] și [2, 7].
- Nodul 5 este **nod terminal** deoarece are suma gradelor egală cu 1 (gradul intern este 1 și gradul extern este 0) și un singur arc incident: [2, 5].
- Nodul 10 este **nod izolat**, deoarece are gradul 0 (niciun arc incident).

Exemplul 2:

G₁₁ Fie graful $G_{11} = (X_{11}, U_{11})$, unde $X_{11} = \{1, 2, 3, 4, 5, 6, 7, 8\}$ și $U_{11} = \{[1, 2], [1, 5], [2, 1], [2, 3], [2, 5], [3, 4], [3, 5], [4, 3], [4, 5], [4, 6], [4, 7], [5, 4]\}$. Din lista arcelor unui graf orientat, puteți preciza următoarele informații:

- **Gradul extern** al unui nod – numărând de câte ori apare eticheta nodului în lista de arce, ca prim element din pereche. Nodul 3 are gradul extern 2 (în mulțimea arcelor, eticheta 3 apare de 2 ori ca prim element: [3, 4] și [3, 5]).
- **Gradul intern** al unui nod – numărând de câte ori apare eticheta nodului în lista de arce, ca al doilea element din pereche. Nodul 5 are gradul 4 (în mulțimea arcelor, eticheta 5 apare de 4 ori ca al doilea element: [1, 5], [2, 5], [3, 5] și [4, 5]).
- **Mulțimea succesorilor unui nod** este formată din nodurile a căror etichetă apare ca al doilea element în perechile în care primul element este nodul precizat. Mulțimea succesorilor nodului 4 este {3, 5, 6, 7} – arcele: [4, 3], [4, 5], [4, 6] și [4, 7].
- **Mulțimea predecesorilor unui nod** este formată din nodurile a căror etichetă apare ca prim element în perechile în care al doilea element este nodul precizat. Mulțimea predecesorilor nodului 3 este {2, 4} – arcele: [2, 3] și [4, 3].

Exemplul 3

G₁₂ Fie graful $G_{12} = (X_{12}, U_{12})$, unde $X_{12} = \{1, 2, 3, 4\}$ și $U_{12} = \{[1, 2], [1, 3], [1, 4], [2, 3], [3, 4], [4, 3]\}$, și **G₁₃** = (X_{13}, U_{13}) , unde $X_{13} = \{1, 2, 3, 4\}$ și $U_{13} = \{[2, 1], [2, 3], [3, 1], [3, 4], [4, 1], [4, 3]\}$. Din lista multilor unui graf, puteți preciza următoarele informații:

- **Nodul sursă** al unui graf apare pe primul loc din pereche de $n-1$ ori – și niciodată pe locul al doilea. În graful G_{12} , nodul 1 este nod sursă. Desenați graful G_{12} pentru a verifica această afirmație.
- **Nodul destinație** al unui graf apare pe al doilea loc din pereche de $n-1$ ori – și niciodată pe primul loc. În graful G_{13} , nodul 1 este nod destinație. Desenați graful G_{13} pentru a verifica această afirmație.

Observație: Într-un graf cu n noduri, oricare ar fi nodul x_k , oricare dintre gradele sale este mai mare sau egal cu 0 și mai mic sau egal cu $n-1$ ($0 \leq d^+(x_k) \leq n-1$ și $0 \leq d^-(x_k) \leq n-1$).

- Temă
:-)
1. În graful G_9 – precizați gradul intern și gradul extern ale nodului 5, identificați nodul cu gradul extern cel mai mare și nodurile cu gradul intern cel mai mic.
 2. În graful G_{10} – identificați nodurile care au gradul intern 1, precizați câte noduri au gradul intern egal cu gradul extern, care sunt nodurile terminale și care sunt nodurile izolate.

3. În graful G_{11} – precizați gradul intern și gradul extern ale nodului 4, identificați nodurile izolate și nodurile terminale, identificați nodurile care au gradul extern maxim și nodurile care au gradul intern egal cu gradul extern.

Teorema 5

Dacă graful orientat G are m arce (u_1, u_2, \dots, u_m) și n noduri (x_1, x_2, \dots, x_n) , atunci între gradele nodurilor și numărul de muchii există următoarea relație: **suma gradelor interne ale tuturor nodurilor este egală cu suma gradelor externe ale tuturor nodurilor și cu numărul de arce:**

$$\sum_{i=1}^n d^+(x_i) = \sum_{i=1}^n d^-(x_i) = m$$

Demonstrație. Fiecare arc $u_k = [x_i, x_j]$ corespunde unei unități din gradul extern al nodului x_i și unei unități din gradul intern al nodului x_j . Rezultă că fiecare arc contribuie cu o unitate la suma gradelor interne și cu o unitate la suma gradelor externe.

Exemplu. În graful G_9 : $d^+(1) + d^+(2) + d^+(3) + d^+(4) + d^+(5) + d^+(6) = 1+3+1+2+2+1 = 10$ și $d^-(1) + d^-(2) + d^-(3) + d^-(4) + d^-(5) + d^-(6) = 1+2+2+2+2+1 = 10$, m fiind egal cu 10.

Temă



În graful G_{11} verificați că este îndeplinită relația dintre gradurile nodurilor și numărul de arce ale grafului.

2.7.4. Reprezentarea și implementarea grafului

Există mai multe moduri de reprezentare la nivel logic a unui graf, care pot fi implementate în memoria unui calculator, folosind diverse tipuri de structuri de date. Aceste reprezentări pot fi folosite în algoritmii care prelucrază grafuri și, implicit, în programele prin care vor fi implementați în calculator acești algoritmi. Printre modurile de reprezentare a unui graf se numără:

- reprezentarea prin **matricea de adiacență**;
- reprezentarea prin **matricea de incidentă**;
- reprezentarea prin **lista muchiilor (arcelor)**;
- reprezentarea prin **lista de adiacență (listele vecinilor)**;
- reprezentarea prin **matricea costurilor**.

Fiecare reprezentare prezintă avantaje în ceea ce privește utilizarea eficientă a memoriei interne, în funcție de tipul grafului (cu noduri puține, dar cu muchii multe – sau cu noduri multe, dar cu muchii puține) și din punct de vedere al eficienței algoritmilor de prelucrare (în funcție de aplicație). În următoarele reprezentări se consideră că graful $G=(X,U)$ are n noduri și m muchii.

2.7.4.1. Reprezentarea prin matricea de adiacență

Matricea de adiacență a unui graf este o matrice pătrată binară de ordinul n ($A_{n,n}$), ale cărei elemente $a_{i,j}$ sunt definite astfel:

$$a_{i,j} = \begin{cases} 1, & \text{dacă } [i, j] \in U \\ 0, & \text{dacă } [i, j] \notin U \end{cases}$$

Implementarea grafului prin matricea de adiacență se face printr-un tablou bidimensional (o matrice pătrată cu dimensiunea n), astfel:

`int a[<n>][<n>];`

Exemple:

Graful neorientat G_1

	1	2	3	4	5	6	7	8
1	0	1	1	1	0	0	0	0
2	1	0	1	0	1	0	0	0
3	1	1	0	1	1	0	0	0
4	1	0	1	0	0	0	0	0
5	0	1	1	0	0	0	0	0
6	0	0	0	0	0	0	1	1
7	0	0	0	0	0	1	0	0
8	0	0	0	0	0	1	0	0

Graful orientat G_9

	1	2	3	4	5	6
1	0	1	0	0	0	0
2	1	0	1	1	0	0
3	0	0	0	0	1	0
4	0	1	0	0	1	0
5	0	0	1	0	0	1
6	0	0	0	0	1	0

Proprietățile matricei de adiacență:

- Elementele de pe diagonala principală au valoarea 0 – din definiția grafului rezultă că orice muchie (arc) $[i, j]$ trebuie să respecte condiția $i \neq j$.
- În cazul unui **graf neorientat**, matricea de adiacență este o matrice simetrică față de diagonala principală, deoarece, dacă există muchia $[i, j]$, atunci există și muchia $[j, i]$.

Această reprezentare este recomandată pentru problemele în care se testează prezența unei muchii sau a unui arc între două noduri, se calculează gradul unui nod etc. – deoarece permite un acces rapid la nodurile și muchiile (arcele) unui graf.

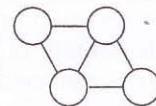
Algoritmi pentru reprezentarea grafurilor folosind matricea de adiacență

Din matricea de adiacență puteți obține următoarele informații:

Graf neorientat	Graf orientat
Suma elementelor matricei de adiacență este egal cu $2 \times m$ (dublul numărului de muchii). Gradul unui nod i este egal cu suma elementelor de pe linia i (sau cu suma elementelor din coloana i).	Suma elementelor matricei de adiacență este egal cu m (numărul de arce). Gradul extern al nodului i este egal cu suma elementelor de pe linia i . Gradul intern al nodului i este egal cu suma elementelor din coloana i .
Nodurile adiacente nodului i sunt nodurile j ($j=1, n$) pentru care elementele din linia i sunt egale cu 1 ($a[i][j]=1$). Mai pot fi definite ca nodurile j ($j=1, n$) pentru care elementele din coloana i sunt egale cu 1 ($a[j][i]=1$).	Succesorii nodului i sunt nodurile j ($j=1, n$) pentru care elementele din linia i sunt egale cu 1 ($a[i][j]=1$). Predecesorii nodului i sunt nodurile j ($j=1, n$) pentru care elementele din coloana i sunt egale cu 1 ($a[j][i]=1$). Nodurile adiacente nodului i sunt nodurile j ($j=1, n$) pentru care elementele din linia i sau din coloana i sunt egale cu 1 ($a[i][j]=1$ sau $a[j][i]=1$) – reuniunea dintre multimea succesorilor și multimea predecesorilor nodului.
Numărul de vecini ai nodului i este egal cu gradul nodului.	Numărul de vecini ai nodului i este egal cu cardinalul mulțimii de noduri adiacente nodului i .
Muchia $[i, j]$ a grafului reprezintă un element al matricei de adiacență care îndeplinește condiția: $a[i][j] = 1$ (sau $a[j][i] = 1$).	Arcul $[i, j]$ al grafului reprezintă un element al matricei de adiacență care îndeplinește condiția: $a[i][j] = 1$

Exemplu

Se consideră graful din figura alăturată. Identificați matricea de adiacență a grafului.



0	1	1	1
1	0	1	0
1	1	0	1
1	0	1	0

0	1	1	1
1	1	1	0
1	1	0	1
1	0	1	0

0	1	1	1
1	0	1	0
1	1	0	0
1	0	1	0

0	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0

(Bacalaureat – Sesiunea specială 2003)

Răspunsul corect este matricea a). Pentru a identifica matricea de adiacență a grafului din figură, se vor elimina pe rând variantele incorecte, prin verificarea următoarelor condiții:

1. Matricea trebuie să fie binară – toate matricele îndeplinesc această condiție;
2. Elementele de pe diagonala principală trebuie să aibă valoarea 0 – matricea b) nu îndeplinește această condiție.
3. Deoarece graful este neorientat, matricea trebuie să fie simetrică – matricea c) nu îndeplinește această condiție.
4. Din analiza grafului se observă că două noduri au gradul 2 și două noduri au gradul 3; în matricea de adiacență trebuie să existe două linii care să contină două elemente cu valoarea 1 și două linii care să contină trei elemente cu valoarea 1 – matricea d) nu îndeplinește această condiție.

Temă

1. Scrieți matricea de adiacență a grafului G_4 . Folosind informațiile din matricea de adiacență, determinați: gradul nodului 5, nodurile izolate și nodurile terminale.
2. Scrieți matricea de adiacență a grafului neorientat $G_{14} = (X_{14}, U_{14})$, unde $X_{14} = \{1, 2, 3, 4\}$ și $U_{14} = \{[1, 2], [2, 3], [3, 4], [4, 1]\}$. Ce proprietate are acest graf?
3. Scrieți matricea de adiacență a grafului G_8 . Folosind informațiile din matricea de adiacență, determinați: gradul intern al nodului 5, gradul extern al nodului 4, succesorii și predecesorii nodului 2 și predecesorii nodului 3.
4. Scrieți matricea de adiacență a grafului G_{11} . Folosind informațiile din matricea de adiacență, determinați: gradul intern al nodului 5, gradul extern al nodului 2, nodurile adiacente nodului 5, succesorii și predecesorii nodului 4, nodurile terminale și nodurile izolate.
5. Scrieți matricea de adiacență a grafului orientat G_{15} din figura 8.
6. Scrieți matricea de adiacență a grafului G_{12} . Cum identificați în matricea de adiacență nodul sursă al grafului?
7. Scrieți matricea de adiacență a grafului G_{13} . Cum identificați în matricea de adiacență nodul destinație al grafului?

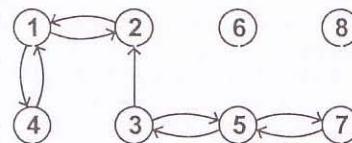


Fig. 8

Implementarea algoritmilor pentru reprezentarea grafurilor cu matricea de adiacență

1. Crearea matricei de adiacență prin introducerea datelor de la tastatură. Determinarea gradului unui nod. Salvarea matricei de adiacență într-un fișier text.

Se citesc de la tastatură muchiile (arcele) unui graf orientat (neorientat), se creează matricea de adiacență a grafului, se afișează nodurile izolate și terminale și se salvează matricea de adiacență în fișierul text *graf1.txt*, pentru graful neorientat, și *graf2.txt*, pentru graful orientat. În fișierul text se scriu, pe primul rând, ordinul grafului, și pe următoarele rânduri, liniile matricei de adiacență. Funcția *scrie()* se folosește pentru a scrie matricea de

adiacentă în fișier. Deoarece matricea de adiacență a fost declarată ca variabilă globală, elementele ei sunt inițializate cu valoarea 0. Pentru testarea programelor se folosesc graful neorientat G_1 și graful orientat G_2 .

Graful neorientat Funcția `grad()` se folosește pentru a determina gradul unui nod.

```
#include<iostream.h>
int a[10][10],n,m;
fstream f("graf1.txt",ios::out);
void scrie()
{int i,j; f<<n<<endl;
 for(i=1;i<=n;i++)
 {for (j=1;j<=n;j++) f<<a[i][j]<<" "; f<<endl;}
 f.close();}
int grad(int i)
{int j,g=0;
 for (j=1;j<=n;j++) g+=a[i][j]; return g;}
void main()
{int i,j,k;
 cout<<"numar de noduri "; cin>>n; cout<<"numar de muchii "; cin>>m;
 for(k=1;k<=m;k++)
 {cout<<"primul nod al muchiei "<<k<<": "; cin>>i;
 cout<<"al doilea nod al muchiei "<<k<<": "; cin>>j;
 a[i][j]=1; a[j][i]=1;}
 cout<<"Nodurile izolate sunt: ";
 for(i=1;i<=n;i++) if (grad(i)==0) cout<<i<<" ";
 cout<<endl<<"Nodurile terminale sunt: ";
 for(i=1;i<=n;i++) if (grad(i)==1) cout<<i<<" ";
 scrie();}
```

Graful orientat Funcția `grad_int()` se folosește pentru a determina gradul intern al unui nod, iar funcția `grad_ext()` se folosește pentru a determina gradul extern al unui nod.

```
#include<iostream.h>
int a[10][10],n,m;
fstream f("graf2.txt",ios::out);
void scrie() //este identică cu cea de la graful neorientat
int grad_ext(int i)
{int j,g=0;
 for (j=1;j<=n;j++) g+=a[i][j]; return g;}
int grad_int(int i)
{int j,g=0;
 for (j=1;j<=n;j++) g+=a[j][i]; return g;}
void main()
{int i,j,k;
 cout<<"numar de noduri "; cin>>n; cout<<"numar de arce "; cin>>m;
 for(k=1;k<=m;k++)
 {cout<<"nodul initial al arcului "<<k<<": "; cin>>i;
 cout<<"nodul final al arcului "<<k<<": "; cin>>j; a[i][j]=1;
 cout<<"Nodurile izolate sunt: ";
 for(i=1;i<=n;i++) if (grad_int(i)+grad_ext(i)==0) cout<<i<<" ";
 cout<<endl<<"Nodurile terminale sunt: ";
 for(i=1;i<=n;i++) if (grad_int(i)+grad_ext(i)==1) cout<<i<<" ";
 scrie();}
```

2. Crearea matricei de adiacență prin citirea datelor din fișier. Determinarea numărului de vecini ai unui nod. Afisarea muchiilor (arcelor) grafului.

Se citesc din fișierele text create anterior matricele de adiacență ale celor două grafuri – *graf1.txt*, respectiv *graf2.txt*. Se afișează nodurile care au cei mai mulți vecini (cele mai multe noduri adiacente). Se determină numărul de muchii (arce) ale grafului și se afișează muchiile (arcelor). Funcția *citeste()* se folosește pentru a citi matricea de adiacență din fișier.

Graful neorientat Funcția *nr_m()* se folosește pentru a determina numărul de muchii ale grafului. La afișarea muchiilor, pentru a nu se afișa de două ori aceeași muchie, se parcurge matricea de adiacență numai deasupra diagonalei principale.

```
#include<iostream.h>
int a[10][10],n;
fstream f("graf1.txt",ios::in);
void citeste()
{int i,j; f>>n;
 for(i=1;i<=n;i++)
   for (j=1;j<=n;j++) f>>a[i][j]; f.close();}
int nr_m()
{int i,j,m=0;
 for(i=1;i<=n;i++)
   for (j=1;j<=n;j++) m+=a[i][j]; return m/2;}
int grad(int i) { // este identică cu cea de la exemplul anterior}
void main()
{int i,j,max; citeste(); max=grad(1);
 for(i=2;i<=n;i++) if (grad(i)>max) max=grad(i);
 cout<<"Nodurile cu cel mai multi vecini sunt: ";
 for(i=1;i<=n;i++) if (grad(i)==max) cout<<i<<" ";
 cout<<endl<<"Graful are "<<nr_m()<<" muchii "<<endl;
 cout<<"Muchiile grafului sunt: ";
 for(i=1;i<=n;i++)
   for (j=i+1;j<=n;j++) if (a[i][j]==1) cout<<i<<" - "<<j<<" ";
```

Graful orientat Funcția *nr_a()* se folosește pentru a determina numărul de arce ale grafului. Funcția *vecini()* se folosește pentru a determina numărul de vecini ai unui nod. La afișarea arcelor, se parcurge toată matricea de adiacență.

```
#include<iostream.h>
int a[10][10],n;
fstream f("graf2.txt",ios::in);
void citeste(){//este identică cu cea de la graful neorientat}
int nr_a()
{int i,j,m=0;
 for(i=1;i<=n;i++)
   for (j=1;j<=n;j++) m+=a[i][j]; return m;}
int vecini(int i)
{int j,v=0;
 for (j=1;j<=n;j++) if (a[i][j]==1 || a[j][i]==1) v++; return v;}
void main()
{int i,j,max; citeste(); max=vecini(1);
 for(i=2;i<=n;i++) if (vecini(i)>max) max=vecini(i);
 cout<<"Nodurile cu cel mai multi vecini sunt: ";
 for(i=1;i<=n;i++) if (vecini(i)==max) cout<<i<<" ";
 cout<<endl<<"Graful are "<<nr_a()<<" arce "<<endl;
```

```
cout<<"Arcele grafului sunt: ";
for(i=1;i<=n;i++)
    for (j=1;j<=n;j++) if (a[i][j]==1) cout<<i<<" - "<<j<<" ";
```

3. Crearea matricei de adiacență prin citirea muchiilor (arcelor) din fișier. Determinarea vecinilor unui nod.

Datele se citesc din fișierul text *graf3.txt*, pentru graful neorientat, și *graf4.txt*, pentru graful orientat. În fișier sunt scrise, pe primul rând, despărțite prin spații, numărul de noduri și numărul de muchii (arce) ale grafului, și apoi pe câte un rând, separate prin spațiu, cele două noduri terminale ale fiecărei muchii (arc). Se afișează vecinii fiecărui nod. Funcția **citeste()** se folosește pentru a citi datele din fișier, iar funcția **vecini()** pentru a determina vecinii unui nod. Fișierele text se creează cu ajutorul unui editor de texte. Pentru testarea programelor se folosesc graful neorientat G_1 și graful orientat G_9 .

Graful neorientat.

```
#include<iostream.h>
int a[10][10],n,m;
fstream f("graf3.txt",ios::in);
void citeste()
{int k,i,j; f>>n>>m;
 for(k=1;k<=m;k++) {f>>i>>j; a[i][j]=1; a[j][i]=1;} f.close();}
void vecini(int i)
{for(int j=1;j<=n;j++) if(a[i][j]==1) cout<<j<<" ";}
void main()
{int i; citeste(); cout<<"Vecinii fiecarui nod sunt: "<<endl;
 for (i=1;i<=n;i++) {cout<<"Nodul "<<i<<": "; vecini(i); cout<<endl;}}
```

Graful orientat.

```
#include<iostream.h>
int a[10][10],n,m;
fstream f("graf4.txt",ios::in);
void citeste()
{int k,i,j; f>>n>>m;
 for(k=1;k<=m;k++) {f>>i>>j; a[i][j]=1;} f.close();}
void vecini(int i)
{for(int j=1;j<=n;j++) if(a[i][j]==1 || a[j][i]==1) cout<<j<<" ";}
void main()
{int i; citeste(); cout<<"Vecinii fiecarui nod sunt: "<<endl;
 for (i=1;i<=n;i++) {cout<<"Nodul "<<i<<": "; vecini(i); cout<<endl;}}
```

4. Generarea matricei de adiacență.

Pentru a testa programele care prelucrează grafuri implementate cu matricea de adiacență, puteți să generați aleatoriu matricea de adiacență. Funcția **generare()** generează matricea de adiacență a unui graf neorientat, cu n noduri și m muchii.

```
#include<iostream.h>
#include<stdlib.h>
int a[10][10],n,m;
void generare()
{int k=0,i,j; randomize();
 while(k<m)
 {i=rand()%n+1; j=rand()%n+1;
 if(i!=j && a[i][j]==0) {a[i][j]=1; a[j][i]=1; k++;}}}
```

```
void main() {cout<<"n= "; cin>>n; cout<<"m= "; cin>>m;
    while (m>n*(n-1)/2) {cout<<"m= "; cin>>m;}
    generare(); ... }
```

Temă

1. Într-un fișier este scrisă o matrice pătrată, astfel: pe primul rând, un număr care reprezintă dimensiunea matricei, și pe următoarele rânduri, valori numerice despărțite prin spațiu – care reprezintă elementele de pe căte o linie a matricei. Să se verifice dacă această matrice poate fi matricea de adiacență a unui graf. În caz afirmativ, să se precizeze dacă graful este orientat sau neorientat.
2. Scrieți un program care să genereze aleatoriu matricea de adiacență a unui graf orientat.
3. Scrieți un program care citește dintr-un fișier matricea de adiacență a unui graf neorientat și care determină numărul minim de muchii care trebuie adăugate pentru ca graful să nu conțină noduri izolate.
4. Scrieți un program care citește, din două fișiere text, g1.txt și g2.txt, matricele de adiacență a două grafuri, $G_a = (X, U_a)$ și $G_b = (X, U_b)$, și care determină matricea de adiacență a **grafului reuniune** $G_r = (X, U_r)$, unde $U_r = U_a \cup U_b$, care se salvează în fișierul g3.txt și matricea de adiacență a **grafului intersecție** $G_i = (X, U_i)$, unde $U_i = U_a \cap U_b$, care se salvează în fișierul g4.txt.
5. Scrieți un program care citește din fișierul text graf2.txt informații despre un graf orientat (de pe prima linie – numărul de noduri, apoi matricea de adiacență) și de la tastatură o mulțime A de numere care reprezintă etichetele unor noduri din graf – și care afișează mulțimea arcelor ce au o extremitate într-un nod din mulțimea A și o extremitate în mulțimea X-A (X fiind mulțimea nodurilor grafului). Pentru testarea programului, se vor folosi graful G_9 și mulțimea $A = \{1, 3, 4, 6\}$.

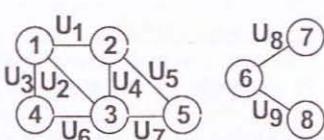
2.7.4.2. Reprezentarea prin matricea de incidentă

Matricea de incidentă a unui graf neorientat este o matrice binară cu n linii și m coloane ($A_{n,m}$), ale cărei elemente $a_{i,j}$ sunt definite astfel:

$$a_{i,j} = \begin{cases} 1, & \text{dacă } [i, j] \in U \\ 0, & \text{dacă } [i, j] \notin U \end{cases}$$

Implementarea grafului neorientat prin matricea de incidentă se face printr-un tablou bidimensional (o matrice cu dimensiunea $n \times m$), astfel:

```
int a[<n>][<m>];
```



Graful G_1
Fig. 9

Graful neorientat G_1

	1	2	3	4	5	6	7	8	9
1	1	1	1	0	0	0	0	0	0
2	1	0	0	1	1	0	0	0	0
3	0	1	0	1	0	1	1	0	0
4	0	0	1	0	0	1	0	0	0
5	0	0	0	0	1	0	1	0	0
6	0	0	0	0	0	0	0	1	1
7	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0	1

Proprietățile matricei de incidentă a grafului neorientat:

1. Pe fiecare coloană există două elemente cu valoarea 1 (pe linile i și j care corespund nodurilor incidente cu muchia), iar restul elementelor au valoarea 0.

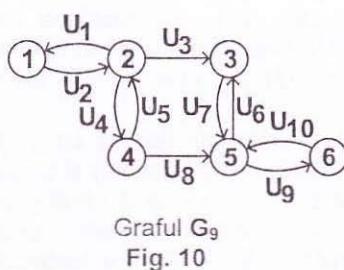
2. Matricea de incidentă are $2 \times m$ elemente egale cu 1, deoarece fiecare muchie este incidentă cu două noduri.

Matricea de incidentă a unui **graf orientat** este o matrice cu n linii și m coloane ($A_{n,m}$), ale cărei elemente $a_{i,j}$ sunt definite astfel:

$$a_{i,j} = \begin{cases} 1, & \text{dacă nodul } i \text{ este extremitatea finală a arcului } j \\ -1, & \text{dacă nodul } i \text{ este extremitatea inițială a arcului } j \\ 0, & \text{dacă nodul } i \text{ nu este extremitatea a arcului } j \end{cases}$$

Implementarea grafului neorientat prin matricea de incidentă se face printr-un tablou bidimensional (o matrice cu dimensiunea $n \times m$), astfel:

```
int a[<n>][<m>];
```



Graful orientat G_9

Graful orientat G_9										
1	2	3	4	5	6	7	8	9	10	
1	-1	0	0	0	0	0	0	0	0	
2	1	-1	-1	-1	1	0	0	0	0	
3	0	0	1	0	0	1	-1	0	0	
4	0	0	0	1	-1	0	0	-1	0	
5	0	0	0	0	0	-1	1	1	1	-1
6	0	0	0	0	0	0	0	0	-1	1

Proprietățile matricei de incidentă a grafului orientat:

- Pe fiecare coloană există un element cu valoarea 1 (pe linia i care corespunde extremității finale a arcului) și un element cu valoarea -1 (pe linia j care corespunde extremității inițiale a arcului), iar restul elementelor au valoarea 0.
- Matricea de incidentă are m elemente egale cu 1 și m elemente egale cu -1, deoarece fiecare arc are o extremitate finală și o extremitate inițială. Suma elementelor matricei este 0.

Această reprezentare este recomandată pentru grafurile care au un număr mare de noduri și un număr mic de muchii.

Algoritmi pentru reprezentarea grafurilor folosind matricea de incidentă

Din matricea de incidentă puteți obține următoarele informații:

Graf neorientat	Graf orientat
Gradul unui nod i este egal cu suma elementelor de pe linia i .	Gradul intern al unui nod i este egal cu numărul de elemente cu valoarea 1 de pe linia i . Gradul extern al unui nod i este egal cu numărul de elemente cu valoarea -1 de pe linia i .
Nodurile adiacente nodului i sunt nodurile j ($j=1,n$) pentru care elementele din linia j sunt egale cu 1 în coloana k ($k=1,m$) în care și elementele de pe linia i au valoarea 1: $a[i][k] = 1$ și $a[j][k] = 1$.	Succesorii nodului i sunt nodurile j ($j=1,n$) pentru care elementele din linia j sunt egale cu 1 în coloana k ($k=1,m$) în care elementele de pe linia i au valoarea -1: $a[i][k] = -1$ și $a[j][k] = 1$. Predecesorii nodului i sunt nodurile j ($j=1,n$) pentru care elementele din linia j sunt egale cu -1 în coloana k ($k=1,m$) în care elementele de pe linia i au valoarea 1: $a[i][k] = 1$ și $a[j][k] = -1$.

Graf neorientat	Graf orientat
	Nodurile adiacente nodului i sunt date de reunirea dintre mulțimea succesorilor și mulțimea predecesorilor nodului.
Numărul de vecini ai nodului i este egal cu gradul nodului.	Numărul de vecini ai nodului i este egal cu cardinalul mulțimii de noduri adiacente nodului i .
Muchia $k = [i,j]$ a grafului este determinată de două elemente ale matricei $a[i][kj]$ și $a[j][kj]$, care îndeplinesc condiția $a[i][k] = 1$ și $a[j][k] = 1$, și semnifică faptul că muchia k este incidentă cu nodurile i și j .	Arcul $k = [i,j]$ al grafului este determinat de două elemente ale matricei, $a[i][kj]$ și $a[j][kj]$, care îndeplinesc condiția $a[i][k] = -1$ și $a[j][k] = 1$, și semnifică faptul că arcul k iese din nodul i și intră în nodul j .

Tema

1. Scrieți matricea de incidentă a grafului neorientat G_4 . Folosind informațiile din matricea de incidentă, determinați: gradul nodului 5, nodurile izolate și nodurile terminale.
2. Scrieți matricea de incidentă a grafului neorientat G_{14} . Ce proprietate are acest graf?
3. Scrieți matricea de incidentă a grafului G_8 . Folosind informațiile din matricea de incidentă, determinați: gradul intern al nodului 5, gradul extern al nodului 4, succesorii și predecesorii nodului 2 și predecesorii nodului 3.
4. Scrieți matricea de incidentă a grafului orientat G_{16} din figura 11.
5. Scrieți matricea de incidentă a grafului G_{11} . Folosind informațiile din matricea de incidentă determinați: gradul intern al nodului 5, gradul extern al nodului 2, nodurile adiacente nodului 5, succesorii și predecesorii nodului 4, nodurile terminale și nodurile izolate.
6. Scrieți matricea de incidentă a grafului G_{13} . Cum identificați, în matricea de incidentă, nodul sursă al unui graf?
7. Scrieți matricea de incidentă a grafului G_{14} . Cum identificați, în matricea de incidentă, nodul destinație al unui graf?

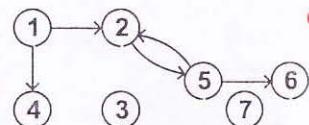
 G_{16} 

Fig. 11

Implementarea algoritmilor pentru reprezentarea grafurilor cu matricea de incidentă

1. Crearea matricei de incidentă prin introducerea datelor de la tastatură. Determinarea gradului unui nod. Salvarea matricei de incidentă într-un fișier text.

Se citesc de la tastatură muchiile (arcele) unui graf orientat (neorientat). Se creează matricea de incidentă a grafului. Se afișează gradul nodului p a cărui etichetă se introduce de la tastatură. Se salvează matricea de incidentă în fișierul text *graf5.txt*, pentru graful neorientat, și *graf6.txt*, pentru graful orientat. În fișierul text se scriu pe primul rând ordinul grafului și numărul de muchii, iar pe următoarele rânduri, liniile matricei de incidentă. Funcția *scrie()* se folosește pentru a scrie matricea de incidentă în fișier. Pentru testarea programelor se folosesc graful neorientat G_1 și graful orientat G_9 .

Graful neorientat. Funcția *grad()* se folosește pentru a determina gradul unui nod.

```
#include<iostream.h>
int a[10][10], n, m;
fstream f("graf5.txt", ios::out);
void scrie()
{
    int i, j; f<<n<<" "<<m<<endl;
    for(i=1; i<=n; i++)
        {for(k=1; k<=m; k++)
            f<<a[i][k]<<" "; f<<endl;}
    f.close();
}
```

```

int grad(int i)
{int g=0,k;
 for(k=1;k<=m;k++) g+=a[i][k]; return g;}
void main()
{int i,j,p,k;
 cout<<"numar de noduri "; cin>>n; cout<<"numar de muchii "; cin>>m;
for(k=1;k<=m;k++)
 {cout<<"primul nod al muchiei "<<k<<": "; cin>>i;
 cout<<"al doilea nod al muchiei "<<k<<": "; cin>>j;
 a[i][k]=1; a[j][k]=1;}
cout<<"nodul= "; cin>>p;
cout<<"Gradul nodului "<<p<<" este "<<grad(p)<<endl;
scrie();}

```

Graful orientat. Funcția `grad_int()` se folosește pentru a determina gradul intern al unui nod, iar funcția `grad_ext()` se folosește pentru a determina gradul extern al unui nod.

```

#include<fstream.h>
int a[10][10],n,m;
fstream f("graf6.txt",ios::out);
int scrie() {//este identică cu cea de la graful neorientat }
int grad_int(int i)
{int g=0,k;
 for(k=1;k<=m;k++) if (a[i][k]==1) g++; return g;}
int grad_ext(int i)
{int g=0,k;
 for(k=1;k<=m;k++) if (a[i][k]==-1) g++; return g;}
void main()
{int i,j,p,k;
 cout<<"numar de noduri "; cin>>n; cout<<"numar de muchii "; cin>>m;
for(k=1;k<=m;k++)
 {cout<<"nodul initial al arcului "<<k<<": "; cin>>i;
 cout<<"nodul final al arcului "<<k<<": "; cin>>j;
 a[i][k]=-1; a[j][k]=1;}
cout<<"nodul= "; cin>>p;
cout<<"Gradul intern al nodului "<<p<<" este "<<grad_int(p)<<endl;
cout<<"Gradul extern al nodului "<<p<<" este "<<grad_ext(p)<<endl;
scrie();}

```

2. Crearea matricei de incidentă prin citirea datelor din fișier. Afisarea vecinilor unui nod.

Se citesc din fișierele create anterior (`graf5.txt`, respectiv `graf6.txt`) matricele de incidentă ale celor două grafuri și se afișează vecinii unui nod x a cărui etichetă se introduce de la tastatură. Funcția `citeste()` se folosește pentru a citi matricea de incidentă din fișier.

Graful neorientat. Funcția `vecini()` se folosește pentru a determina vecinii unui nod.

```

#include<fstream.h>
int a[10][10];
fstream f("graf5.txt",ios::in);
void citeste()
{int i,k; f>>n>>m;
for(i=1;i<=n;i++)
 for (k=1;k<=m;k++) f>>a[i][k]; f.close();}

```

```

void vecini(int i)
{int k,j;
 for(k=1;k<=m;k++)
 if(a[i][k]==1) for(j=1;j<=n;j++)
 if(j!=i && a[j][k]==1) cout<<j<<" ";
void main()
{int p; cout<<"nodul= "; cin>>x; citeste();
 cout<<"Vecinii nodului "<<x<<" sunt nodurile "; vecini(x);}

```

Graful orientat Vectorii binari s și p se folosesc pentru a memora succesorii, respectiv predecesorii nodului x . Elementul i are valoarea 1 dacă nodul i este succesor, respectiv predecesor al nodului x ; altfel, are valoarea 0. Funcțiile `succ()` și `pred()` se folosesc pentru a determina în vectorii s și p succesorii, respectiv predecesorii unui nod. Funcția `vecini()` se folosește pentru a determina vecinii unui nod, prin reuniunea mulțimii predecesorilor și a succesorilor.

```

#include<iostream.h>
int a[10][10],n,m,s[10],p[10];
fstream f("graf6.txt",ios::in);
void citeste() { //este identică cu cea de la graful neorientat }
void succ(int i)
{for(int k=1;k<=m;k++)
 if(a[i][k]==-1) for(int j=1;j<=n;j++)
 if(j!=i && a[j][k]==1) s[j]=1;}
void pred(int i)
{for(int k=1;k<=m;k++)
 if(a[i][k]==1) for(int j=1;j<=n;j++)
 if(j!=i && a[j][k]==-1) p[j]=1;}
void vecini(int i)
{int j; succ(i); pred(i);
 for(j=1;j<=n;j++) if (j!=i && (s[j]==1 || p[j]==1)) cout<<j<<" ";}
void main()
{int x; cout<<"nodul= "; cin>>x; citeste();
 cout<<"Vecinii nodului "<<x<<" sunt nodurile "; vecini(x);}

```

3. Crearea matricei de incidență prin citirea muchiilor (arcelor) din fișier. Prelucrarea informațiilor asociate muchiilor.

Datele se citesc din fișierul text *graf7.txt*, pentru graful neorientat, și *graf8.txt*, pentru graful orientat. În fișier sunt scrise, pe primul rând, despărțite prin spații, numărul de noduri și numărul de muchii (arce) ale grafului, și apoi, pe câte un rând, separate prin spațiu, cele două noduri terminale ale fiecărei muchii (arc) și lungimea muchiei (arcului). Se afișează muchiile (arcele) care au lungimea mai mare decât lungimea medie a muchiilor (arcelor) din graf. Se folosește vectorul d pentru a memora lungimea fiecărei muchii (arc). Funcția `citeste()` se folosește pentru citi datele din fișier, funcția `medie()` pentru a determina lungimea medie a muchiilor (arceor) din graf iar funcția `afisare()` pentru a afișa muchiile (arcele) care au lungimea mai mare decât media. Fișierele text se creează cu ajutorul unui editor de texte. Pentru testarea programelor se folosesc graful neorientat G_1 și graful orientat G_9 . În aceste grafuri se asociază fiecărei muchii (arc) o valoare pentru lungime.

Graful neorientat

```

#include<iostream.h>
int a[10][10],d[10],n,m;
fstream f("graf7.txt",ios::in);

```

```

void citeste()
{int k,i,j,l; f>>n>>m;
 for(k=1;k<=m;k++) {f>>i>>j>>l; a[i][k]=1; a[j][k]=1; d[k]=1;}
 f.close();}
float media()
{int i,s=0;
 for(i=1;i<=m;i++) s+=d[i]; return (float)s/m;}
void afiseaza()
{int i,k; float dm=media();
 for(k=1;k<=m;k++)
 if (d[k]>dm)
 {for (i=1;i<=n;i++) if (a[i][k]==1) cout<<i<<" ";
 cout<<" cu lungimea "<<d[k]<<endl;}}
void main()
{citeste(); cout<<"Media lungimii este: "<<media()<<endl;
 cout<<"Muchiile sunt: "<<endl; afiseaza();}

```

Graful orientat

```

#include<iostream.h>
int a[10][15],d[15],n,m;
fstream f("graf8.txt",ios::in);
void citeste()
{int k,i,j,l; f>>n>>m;
 for(k=1;k<=m;k++) {f>>i>>j>>l; a[i][k]=-1; a[j][k]=1; d[k]=1;}
 f.close();}
float media() //este identică cu cea de la graful neorientat }
void afiseaza()
{int i,k,x,y; float dm=media();
 for(k=1;k<=m;k++)
 if (d[k]>dm)
 {for (i=1;i<=n;i++) {if (a[i][k]==-1) x=i;
 if (a[i][k]==1) y=i;}
 cout<<x<<" - "<<y<<" cu lungimea "<<d[k]<<endl;}}
void main()
{citeste(); cout<<"Media lungimii este: "<<media()<<endl;
 cout<<"Arcele sunt: "<<endl; afiseaza();}

```

4. Crearea matricei de incidentă prin citirea datelor din matricea de adiacență. Afișarea muchiilor și a nodurilor izolate.

Matricele de adiacență se citesc din fișierele text create anterior: *graf1.txt*, pentru graful neorientat, și *graf2.txt*, pentru graful orientat. Se creează matricele de incidentă ale celor două grafuri, se afișează muchiile (arcele) și nodurile izolate. Se salvează în fișierul text *graf9.txt*, respectiv *graf10.txt*, informațiile despre muchii, astfel: pe primul rând, despărțite prin spații, numărul de noduri și numărul de muchii (arce) ale grafului, și apoi, pe câte un rând, separate prin spațiu, cele două noduri terminale ale fiecărei muchii (arc). Se folosesc matricele a pentru matricea de adiacență și b pentru matricea de incidentă și funcțiile *citeste()* pentru a citi matricea de adiacență din fișier, *salveaza()* pentru a salva în fișierul text informațiile despre muchiile (arcele) grafului, *transforma()* pentru a obține matricea de incidentă din matricea de adiacență, *afiseaza_noduri_izolate()* pentru a afișa nodurile izolate, și *afiseaza_muchii()* pentru a afișa muchiile (arcele). Pentru testarea programelor se folosesc graful neorientat G₁ și graful orientat G₉.

Graful neorientat

```
#include<fstream.h>
int a[10][10],b[10][20],n,m;
fstream f1("graf1.txt",ios::in),f2("graf9.txt",ios::out);
void citeste()
{int i,j; f1>>n;
 for(i=1;i<=n;i++)
  for(j=1;j<=n;j++) {f1>>a[i][j]; if(a[i][j]==1) m++;}
 m=m/2; f1.close();}
void transforma()
{int i,j,k=1;
 for (i=1;i<=n;i++)
  for (j=1;j<i;j++) if (a[i][j]==1) {b[i][k]=1; b[j][k]=1; k++;}
void afiseaza_muchii()
{for (int k=1; k<=m; k++)
 {cout<<"Muchia "<<k<<": ";
  for (int i=1;i<=n;i++) if (b[i][k]==1) cout<<i<<" "; cout<<endl;}
void afiseaza_noduri_izolate()
{int i,k,x;
 for (i=1;i<=n;i++)
  {for (k=1,x=0;k<=m && x==0;k++) if (b[i][k]==1) x=1;
   if (!x) cout<<i<<" ";}
void salveaza()
{f2<<n<<" "<<m<<endl;
 for (int k=1;k<=m;k++)
  {for (int i=1;i<=n;i++) if (b[i][k]==1) f2<<i<<" "; f2<<endl;}
 f2.close();}
void main()
{citeste(); transforma(); afiseaza_muchii();
 cout<<"Nodurile izolate sunt: "; afiseaza_noduri_izolate();
 salveaza();}
```

Graful orientat

```
#include<fstream.h>
int a[10][10],b[10][20],n,m;
fstream f1("graf1.txt",ios::in),f2("graf9.txt",ios::out);
void citeste()
{int i,j; f1>>n;
 for(i=1;i<=n;i++)
  for(j=1;j<=n;j++) {f1>>a[i][j]; if(a[i][j]==1) m++;} f1.close();}
void transforma()
{int i,j,k=1;
 for (i=1;i<=n;i++)
  for (j=1;j<=n;j++) if (a[i][j]==1) {b[i][k]=-1; b[j][k]=1; k++;}
void afiseaza_arce()
{int i,k,x,y;
 for (k=1; k<=m; k++)
 {cout<<"Arcul "<<k<<": ";
  for (i=1;i<=n;i++) {if (b[i][k]==-1) x=i; if (b[i][k]==1) y=i;}
  cout<<x<<"-<<y<<endl;}}
void afiseaza_noduri_izolate()
{int i,k,x;
```

```

for (i=1;i<=n;i++)
{for (k=1,x=0;k<=m && x==0;k++) if (b[i][k]==1 || b[i][k]==-1) x=1;
 if (!x) cout<<i<<" ;"}
void salveaza()
{int i,k;
f2<<n<<" "<<m<<endl;
for (k=1;k<=m;k++)
{for (i=1;i<=n;i++) {if (b[i][k]==-1) x=i; if (b[i][k]==1) y=i;}
f2<<x<<" "<<y<<endl;}
f2.close();}
void main()
{citeste(); transforma(); afiseaza_arce();
cout<<"Nodurile izolate sunt: "; afiseaza_noduri_izolate();
salveaza();}

```

Temă

- Într-un fișier text este scrisă o matrice, astfel: pe primul rând – două numere separate prin spațiu, care reprezintă numărul de linii și numărul de coloane ale matricei, și, pe următoarele rânduri – valori numerice despărțite prin spațiu, care reprezintă elementele de pe căte o linie a matricei.
 - Scriți un program care să verifice dacă această matrice poate fi matricea de incidentă a unui graf neorientat. În caz afirmativ, să se afișeze căte noduri izolate are graful (**Indicație**. Se verifică următoarele condiții: a) să fie o matrice binară; b) suma elementelor de pe fiecare coloană să fie egală cu 2; c) să nu existe două coloane identice. Un nod este izolat dacă suma elementelor de pe linia sa este egală cu 0).
 - Scriți un program care să verifice dacă această matrice poate fi matricea de incidentă a unui graf orientat. În caz afirmativ, să se determine căte noduri care au gradul intern egal cu gradul extern există (**Indicație**. Se verifică următoarele condiții: a) pe fiecare coloană să nu existe decât o valoare egală cu 1, una egală cu -1 și restul egale cu 0; b) să nu existe două coloane identice. Un nod are gradul intern egal cu gradul extern dacă suma elementelor de pe linia sa este egală cu 0).
- Scriți un program care citește din fișierul text *graf6.txt* matricea de incidentă a grafului orientat și care:
 - afișează numărul de vecini ai unui nod *p* a cărui etichetă se citește de la tastatură;
 - generează matricea de adiacență a grafului din matricea de incidentă și o salvează în fișierul *graf6a.txt*.

2.7.4.3. Reprezentarea prin lista muchiilor

Lista muchiilor unui graf este formată din *m* elemente care conțin, fiecare, căte o pereche de două noduri, *x_i* și *x_j*, care formează o muchie, adică pentru care $[x_i, x_j] \in U$.

Implementarea acestui tip de reprezentare se poate face folosind una dintre următoarele structuri de date:

- **Matricea muchiilor *u* cu dimensiune *mx2***, în care fiecare linie corespunde unei muchii (arc) și în fiecare linie se înregistrează în cele două coloane etichetele nodurilor care se găsesc la extremitățile muchiei (arcului).

```
int u [<m>][2];
```

Referirea la nodurile adiacente muchiei (arcului) *i* se face prin *u[i][0]* – extremitatea inițială a muchiei (arcului), respectiv *u[i][1]* – extremitatea finală a muchiei (arcului).

→ **Vectorul muchiilor u** cu dimensiunea m ale cărui elemente sunt înregistrări, fiecare înregistrare fiind formată din două câmpuri x și y ce conțin etichetele nodurilor care se găsesc la extremitățile muchiei (arcului). Pentru elementele vectorului se definește tipul de date ***muchie***, de tip înregistrare.

```
struct muchie {int x,y;};
muchie u[<m>];
```

Referirea la o muchie (arc) i se face prin $u[i]$, iar la unul dintre nodurile adiacente muchiei (arcului) se face prin $u[i].x$ – extremitatea inițială a muchie (arcului), respectiv $u[i].y$ – extremitatea finală a muchiei (arcului).

Implementarea cu matrice

	0	1
1	1	2
2	1	3
3	1	4
4	2	3
5	2	4
6	3	4
7	3	5
8	6	7
9	6	8

Graful neorientat G_1

Implementarea cu vector de înregistrări

Muchiile

1	2	3	4	5	6	7	8	9
1	2	1	3	2	4	2	3	2

Implementarea cu matrice

	0	1
1	1	2
2	2	1
3	2	4
4	3	2
5	3	5
6	4	2
7	4	5
8	5	3
9	5	6
10	6	5

Graful orientat G_2

Implementarea cu vector de înregistrări

Arcele

1	2	2	1	2	4	3	2	3	5
1	2	2	1	2	4	3	2	4	5

Exemple:

Dacă implementarea se face folosind matricea, atunci pentru orice muchie (arc) i , $u[i][0] \neq u[i][1]$. Dacă implementarea se face folosind vectorul de înregistrări, atunci pentru orice muchie (arc) i , $u[i].x \neq u[i].y$.

Această reprezentare este recomandată pentru problemele în care se face prelucrarea succesivă a muchiilor (arcelor). Are avantajul că permite adăugarea la tipul de date ***muchie*** și a altor câmpuri (lungime, cost, timp etc.), corespunzător unor mărimi care pot fi asociate muchiilor (arcelor).

Algoritmi pentru reprezentarea grafurilor folosind lista muchiilor

Din lista muchiilor puteți obține următoarele informații:

Graf neorientat	Graf orientat
Gradul unui nod <i>i</i> este egal, în funcție de implementare, cu numărul de apariții ale etichetei nodului în matrice, respectiv în câmpurile vectorului de înregistrări.	Gradul extern al nodului <i>i</i> este egal, în funcție de implementare, cu numărul de apariții ale etichetei nodului în coloana 0 a matricei, respectiv în primul câmp, în vectorul de înregistrări. Gradul intern al nodului <i>i</i> este egal, în funcție de implementare, cu numărul de apariții ale etichetei nodului în coloana 1 a matricei, respectiv în al doilea câmp, în vectorul de înregistrări.
Nodurile adiacente nodului <i>i</i> sunt, în funcție de implementare, etichetele <i>j</i> din coloana 1, pentru care $u[k][0]=i$, sau din coloana 0, pentru care $u[k][1]=i$, respectiv etichetele <i>j</i> din câmpul $u[k].y$, pentru care $u[k].x=i$, sau din câmpul $u[k].x$, pentru care $u[k].y=i$ ($k=1, m$).	Succesorii nodului <i>i</i> sunt, în funcție de implementare, etichetele <i>j</i> din coloana 1 pentru care $u[k][0]=i$, respectiv etichetele <i>j</i> din câmpul $u[k].y$ pentru care $u[k].x=i$ ($k=1, m$). Predecesorii nodului <i>i</i> sunt, în funcție de implementare, etichetele <i>j</i> din coloana 0 pentru care $u[k][1]=i$, respectiv etichetele <i>j</i> din câmpul $u[k].x$ pentru care $u[k].y=i$ ($k=1, m$). Nodurile adiacente nodului <i>i</i> sunt date de reunirea dintre mulțimea succesorilor și mulțimea predecesorilor nodului.
Numărul de vecini ai nodului <i>i</i> este egal cu gradul nodului.	Numărul de vecini ai nodului <i>i</i> este egal cu cardinalul mulțimii de noduri adiacente nodului <i>i</i> .

Tema

1. Scrieți lista muchiilor a grafului G_4 . Folosind informațiile din lista muchiilor, determinați: gradul nodului 5, nodurile izolate și nodurile terminale.
2. Scrieți lista muchiilor a grafului neorientat G_{14} . Ce proprietate are acest graf?
3. Scrieți lista muchiilor a grafului G_8 . Folosind informațiile din lista muchiilor, determinați: gradul intern al nodului 5, gradul extern al nodului 4, succesorii și predecesorii nodului 2 și predecesorii nodului 3.
4. Scrieți lista muchiilor a grafului G_{11} . Folosind informațiile din lista muchiilor, determinați: gradul intern al nodului 5, gradul extern al nodului 2, nodurile adiacente nodului 5, succesorii și predecesorii nodului 4, nodurile terminale și nodurile izolate.
5. Scrieți lista muchiilor a grafului orientat G_{17} din figura 12. Folosind informațiile din lista muchiilor, identificați nodurile izolate.
6. Scrieți lista muchiilor a grafului G_{13} . Cum identificați, în lista muchiilor, nodul sursă?
7. Scrieți lista muchiilor a grafului G_{14} . Cum identificați, în lista muchiilor, nodul destinație?

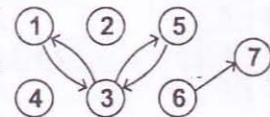


Fig. 12

Implementarea algoritmilor pentru reprezentarea grafurilor cu lista muchiilor

1. Crearea matricei cu lista muchiilor prin introducerea datelor de la tastatură. Determinarea gradului unui nod. Salvarea informațiilor despre muchii într-un fișier text.

Se citesc de la tastatură muchiile (arcele) unui graf orientat (neorientat). Se creează matricea cu muchiile grafului. Se afișează nodurile izolate și terminale. Se salvează matricea cu muchiile grafului în fișierul text *graf11.txt*, pentru graful neorientat, și *graf12.txt*, pentru graful orientat. În fișierul text se vor scrie, pe primul rând, ordinul grafului și numărul de muchii, iar

pe următoarele rânduri, nodurile de la extremitățile unei muchii (arc). Funcția `scrie()` se folosește pentru a scrie informațiile din matricea muchiilor în fișier. Pentru testarea programelor se folosesc graful neorientat G_1 și graful orientat G_9 .

Graful neorientat. Funcția `grad()` se folosește pentru a determina gradul unui nod.

```
#include<iostream.h>
int a[10][2],n,m;
fstream f("graf11.txt",ios::out);
int grad(int i)
{int k,g=0;
 for (k=1;k<=m;k++) if (a[k][0]==i || a[k][1]==i) g++; return g;}
void scrie()
{int k; f<<n<<" "<<m<<endl;
 for(k=1;k<=m;k++) f<<a[k][0]<<" "<<a[k][1]<<endl; f.close();}
void main()
{int i,j,k;
 cout<<"numar de noduri "; cin>>n; cout<<"numar de muchii "; cin>>m;
 for(k=1;k<=m;k++)
 {cout<<"primul nod al muchiei "<<k<<": "; cin>>i;
 cout<<"al doilea nod al muchiei "<<k<<": "; cin>>j;
 a[k][0]=i; a[k][1]=j;}
 cout<<"Nodurile izolate sunt: ";
 for(i=1;i<=n;i++) if (grad(i)==0) cout<<i<<" ";
 cout<<endl<<"Nodurile terminale sunt: ";
 for(i=1;i<=n;i++) if (grad(i)==1) cout<<i<<" ";
 scrie();}
```

Graful orientat. Funcția `grad_int()` se folosește pentru a determina gradul intern al unui nod, iar funcția `grad_ext()` se folosește pentru a determina gradul extern al unui nod.

```
#include<iostream.h>
int a[10][2],n,m;
fstream f("graf12.txt",ios::out);
int grad_int(int i)
{int g=0,k;
 for(k=1;k<=m;k++) if (a[k][1]==i) g++; return g;}
int grad_ext(int i)
{int g=0,k;
 for(k=1;k<=m;k++) if (a[k][0]==i) g++; return g;}
void scrie() { //este identică cu cea de la graful neorientat }
void main()
{int i,j,k;
 cout<<"numar de noduri "; cin>>n; cout<<"numar de arce "; cin>>m;
 for(k=1;k<=m;k++)
 {cout<<"nodul initial al arcului "<<k<<": "; cin>>i;
 cout<<"nodul final al arcului "<<k<<": "; cin>>j;
 a[k][0]=i; a[k][1]=j;}
 cout<<"Nodurile izolate sunt: ";
 for(i=1;i<=n;i++) if (grad_int(i)+grad_ext(i)==0) cout<<i<<" ";
 cout<<endl<<"Nodurile terminale sunt: ";
 for(i=1;i<=n;i++) if (grad_int(i)+grad_ext(i)==1) cout<<i<<" ";
 scrie();}
```

2. Crearea vectorului de muchii prin citirea muchiilor (arcelor) din fișier. Prelucrarea informațiilor asociate muchiilor.

Datele se citesc din fișierele text create anterior: *graf7.txt*, pentru graful neorientat, și *graf8.txt*, pentru graful orientat. Pentru un nod *p* a cărui etichetă se citește de la tastatură, se afișează cel mai apropiat vecin (sau cei mai apropiati vecini, dacă există mai mulți) la care se poate ajunge din nodul *p*. În cazul grafului neorientat, cel mai apropiat vecin este nodul adiacent nodului *p* care formează cu acesta muchia care are lungimea cea mai mică față de muchiile incidente cu ceilalți vecini. În cazul grafului orientat, cel mai apropiat vecin la care se poate ajunge din nodul *p* este nodul succesor nodului *p* care formează cu acesta arcul care are lungimea cea mai mică față de arcele incidente cu ceilalți succesorii. Funcția *citeste()* se folosește pentru a citi datele din fișier. Pentru testarea programelor se folosesc graful neorientat *G₁* și graful orientat *G₉*.

Graful neorientat. Funcția *izolat()* determină dacă nodul este izolat.

```
#include<iostream.h>
struct muchie {int x,y,d;};
muchie u[20]; int n,m;
fstream f("graf7.txt",ios::in);
void citeste()
{int k; f>>n>>m;
 for(k=1;k<=m;k++) f>>u[k].x>>u[k].y>>u[k].d; f.close();}
int izolat(int i)
{int k,g=0;
 for (k=1;k<=m;k++) if (u[k].x==i || u[k].y==i) g++; return g==0;}
void main()
{int k,p,min;
 citeste(); cout<<"Nodul: "; cin>>p;
 if (izolat(p)) cout<<"Nodul "<<p<<" nu are vecini";
 else
 {k=1;
 while (u[k].x!=p && u[k].y!=p) k++;
 min=u[k].d;
 for (k++;k<=m;k++)
 {if (u[k].x==p || u[k].y==p)
 if (u[k].d<min) min=u[k].d;
 cout<<"Distanta minima este "<<min<<endl;
 cout<<"Nodurile aflate la distanta minima sunt: ";
 for (k=1;k<=m;k++)
 {if (u[k].x==p && u[k].d==min) cout<<u[k].y<<" ";
 if (u[k].y==p && u[k].d==min) cout<<u[k].x<<" ";}}}}
```

Graful orientat. Funcția *succ()* determină dacă nodul are succesorii.

```
#include<iostream.h>
struct arc {int x,y,d;};
arc u[20];
int n,m;
fstream f("graf8.txt",ios::in);
void citeste() {//este identică cu cea de la graful neorientat }
int succ(int i)
{int k,g=0;
 for (k=1;k<=m;k++) if (u[k].x==i) g++; return g!=0;}}
```

```

void main()
{
int k,p,min; citeste(); cout<<"Nodul: "; cin>>p;
if (!succ(p))
    cout<<"Nodul "<<p<<" nu are vecini la care sa se poata ajunge";
else
{k=1;
while (u[k].x!=p) k++;
min=u[k].d;
for (k++;k<=m;k++) if (u[k].x==p && u[k].d<min) min=u[k].d;
cout<<"Distanta minima este "<<min<<endl;
cout<<"Nodurile aflate la distanta minima sunt: ";
for (k=1;k<=m;k++)
    if (u[k].x==p && u[k].d==min) cout<<u[k].y<< " ";
}

```

Temă

1. Scrieți un program care realizează următoarele:

- reface matricea cu muchiile grafului orientat din fișierul text *graf12.txt*,
- determină câte noduri izolate are graful și afișează aceste noduri;
- generează matricea de adiacență din matricea muchiilor și o salvează în fișierul text *graf12a.txt*.

(Indicație. Se numără nodurile distincte care apar în lista muchiilor – **n1**, iar numărul de noduri izolate va fi dat de diferența **n-n1**.)

- Scrieți un program care construiește vectorul de muchii din matricea de adiacență a grafului orientat care se citește din fișierul text *graf2.txt* și care determină vecinii unui nod **p** a cărui etichetă se citește de la tastatură.
- Scrieți un program care să genereze aleatoriu, într-un fișier, lista muchiilor unui graf neorientat (orientat). Numărul de noduri și numărul de muchii se citesc de la tastatură.
- Scrieți un program care să genereze aleatoriu, într-un fișier, lista muchiilor unui graf neorientat (orientat) în care muchiile au asociate o mărime numită cost. Se citesc de la tastatură: numărul de noduri, numărul de muchii și limitele intervalului în care mărimea cost poate lua valori.

2.7.4.4. Reprezentarea prin lista de adiacență

Lista de adiacență este formată din listele L_i ($1 \leq i \leq n$) care conțin toti vecinii unui nod x_i la care se poate ajunge direct din nodul x_i , adică toate nodurile x_j pentru care $[x_i, x_j] \in U$.

Observație. În cazul grafului neorientat, lista L_i a vecinilor unui nod x_i al grafului este formată din nodurile x_j adiacente nodului x_i . În cazul grafului orientat, lista L_i a vecinilor unui nod x_i al grafului este formată din nodurile x_j care sunt succesorii nodului x_i .

Implementarea acestui tip de reprezentare se poate face:

→ **static**, folosind una dintre următoarele structuri de date:

- Matricea liste de adiacență
- Vectorii liste de adiacență

→ **dinamic**, cu ajutorul listelor înlățuite.

Implementarea statică

A. **Matricea liste de adiacență L** cu 2 linii și $n+2 \times m$ coloane pentru graful neorientat, respectiv cu $n+m$ coloane pentru graful orientat, definită astfel:

Graful neorientat G_1

Nod	Lista de adiacență
1	2, 3, 4
2	1, 3, 5
3	1, 2, 4, 5
4	1, 3
5	2, 3
6	7, 8
7	6
8	6

- Prima linie conține etichetele nodurilor și liste de adiacență ale fiecărui nod; este formată din două secțiuni:

a. Primele n coloane conțin etichetele nodurilor:

$$L[0][i]=i \quad (1 \leq i \leq n)$$

b. Următoarele $m \times 2$ coloane, respectiv m coloane, conțin în ordine cele n liste de adiacență ale celor n noduri.

- A doua linie conține informațiile necesare pentru a identifica în prima linie lista de adiacență a fiecărui nod; este formată din două secțiuni:

a. Primele n coloane conțin, în ordine, pentru fiecare nod i ($1 \leq i \leq n$), indicele coloanei din prima linie din care începe lista de adiacență a nodului: $L[1][i]=j$, unde j este indicele coloanei de unde începe lista de adiacență a nodului i . Dacă nodul este izolat, se va memora valoarea 0 – $L[1][i]=0$ (nu există listă de adiacență pentru acel nod).

b. Următoarele $m \times 2$ coloane, respectiv m coloane, conțin în ordine informații despre modul în care se înlăntăresc elementele din listă. Dacă nodul $L[0][i]$ se găsește în interiorul listei, atunci $L[1][i]=i+1$ (indicele următorului element din listă). Dacă nodul $L[0][i]$ se găsește la sfârșitul listei, atunci $L[1][i]=0$ (s-a terminat lista de adiacență a nodului i).

Matricea este definită astfel:

a) pentru graful neorientat: `int L[2][<n>+2*<m>];`

b) pentru graful orientat: `int L[2][<n>+<m>];`

Graful neorientat G_1

Nodurile								L ₁		L ₂			L ₃			L ₄		L ₅		L ₆		L ₇		L ₈	
1	2	3	4	5	6	7	8	2	3	4	1	3	5	1	2	4	5	1	3	2	3	7	8	6	6
9	12	15	19	21	23	25	26	10	11	0	13	14	0	16	17	18	0	20	0	22	0	24	0	0	0

Graful orientat G_9

Nodurile								L ₁		L ₂		L ₃		L ₄		L ₅		L ₆		
1	2	3	4	5	6	2	7	1	3	4	5	2	5	3	6	8				
7	8	11	12	14	16	0	9	10	0	0	20	0	22	0	0					

Indicii coloanelor

Indicii coloanelor

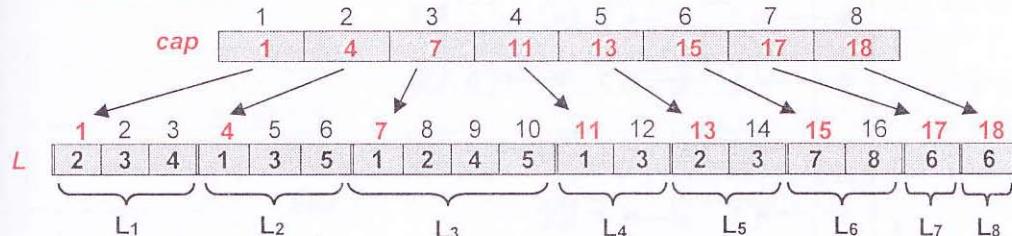
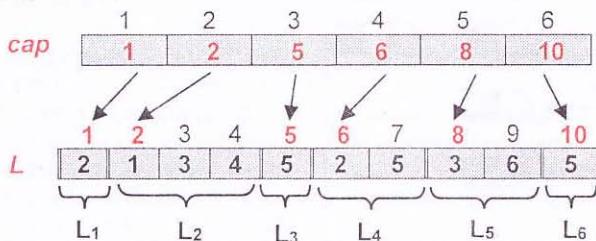
- B. **Vectorii liste de adiacență:** un vector L cu dimensiunea $m \times 2$, pentru graful neorientat, respectiv cu dimensiunea m , pentru graful orientat, care conține liste de adiacență ale fiecărui nod, și un vector cap , cu dimensiunea n , care conține indicii de la care începe lista vecinilor fiecărui nod în vectorul L . Indicii din vectorul cap corespund etichetelor nodurilor. Cei doi vectori sunt definiți astfel:

a) pentru graful neorientat: `int cap[<n>], L[2*<m>];`

b) pentru graful orientat: `int cap[<n>], L[<m>];`

Graful orientat G_9

Nod	Listă de adiacență
1	2
2	1, 3, 4
3	5
4	2, 5
5	3, 6
6	5

Graful neorientat G_1 **Graful orientat G_9** 

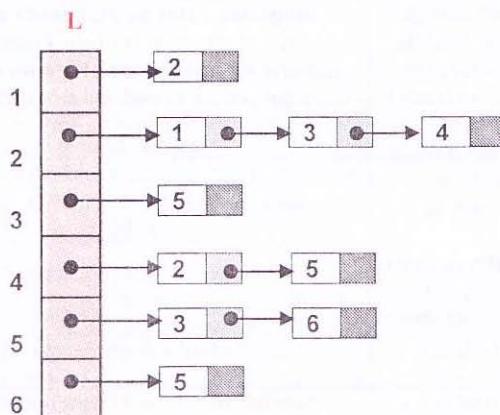
Observație. Fiecare listă de vecini L_i , conține indicii coloanelor j în care se găsesc valori de 1 în matricea de adiacență ($a[i][j]=1$).

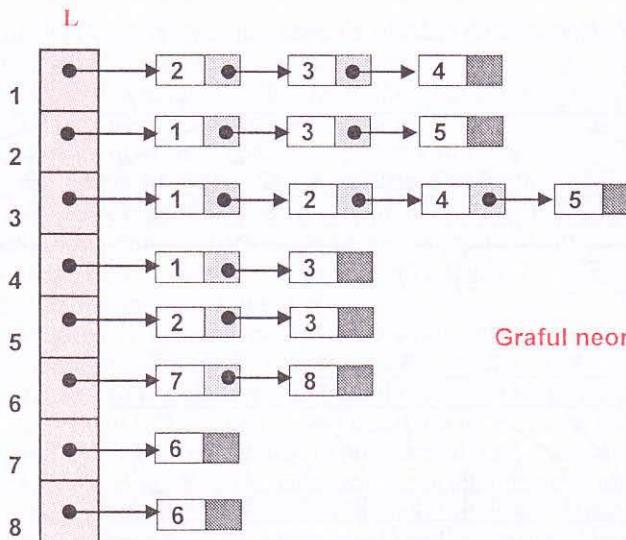
Implementarea dinamică

Lista de adiacență a fiecărui nod se memorează într-o listă simplu înlățuită, ale cărei elemente sunt de tip **nod** (informația utilă memorează eticheta unui nod din listă), iar adresa primului element din fiecare listă se memorează într-un vector **L**, care are lungimea logică egală cu numărul de noduri **n** și ale căruia elemente sunt de tip **pointer** către tipul **nod**:

```
struct nod {int info;
            nod * urm;};

nod *L[<n>];
```

**Graful orientat G_9**

**Graful neorientat G_1**

Această reprezentare este recomandată pentru grafurile care au un număr mare de noduri și un număr mic de muchii.

Algoritmi pentru reprezentarea grafurilor folosind listele de adiacență

Din lista de adiacență implementată static cu matrice puteți obține următoarele informații:

Graf neorientat	Graf orientat
Lungimea listei de adiacență a nodului i neizolat, cu eticheta mai mică decât n , este egală cu diferența dintre primul indice j ($j=i+1, n-1$) diferit de 0, din prima secțiune a liniei a doua ($L[1][j] \neq 0$), și indicele coloanei din care începe lista de adiacență a nodului i ($L[1][j] - L[1][i]$). Pentru nodul n , lungimea listei de adiacență este egală cu diferența dintre numărul total de coloane, plus 1, și indicele coloanei din care începe lista de adiacență a nodului n ($n+2 \times m + 1 - L[1][n]$). Lungimea listei de adiacență se mai poate determina prin numărarea în linia a două a elementelor diferite de 0, începând cu elementul din coloana $L[1][i]$), la care se adaugă 1.	Lungimea listei de adiacență a nodului i neizolat, cu eticheta mai mică decât n , se calculează la fel ca și în cazul grafului neorientat. Pentru nodul n lungimea listei de adiacență este egală cu $(n+m+1 - L[1][n])$. Gradul extern al nodului i este egal cu lungimea listei de adiacență a nodului. Gradul intern al nodului i este egal, cu numărul de aparitii ale etichetei nodului i în a doua secțiune a primei linii a matricei ($L[0][j]=i$, cu $j=n+1, n+m$).
Gradul unui nod i este egal cu lungimea listei de adiacență a nodului sau cu numărul de aparitii ale etichetei nodului i în a doua secțiune a primei linii a matricei ($L[0][j]=i$, cu $j=n+1, n+2 \times m$).	
Nodurile adiacente nodului i sunt nodurile a căror etichetă apare în lista de adiacență a nodului i .	Succesorii nodului i sunt nodurile a căror etichetă apare în lista de adiacență a nodului i . Predecesorii nodului i sunt nodurile j în a căror listă de adiacență apare nodul i .

Graf neorientat	Graf orientat
<p>Numărul de vecini ai nodului i este egal cu gradul nodului.</p> <p>Muchia $[i,j]$ a grafului reprezintă nodul i și un nod j din lista de adiacență a nodului i din prima linie a matricei $(L[0][j] \in L_i)$.</p>	<p>Nodurile adiacente nodului i sunt date de reunirea dintre mulțimea succesorilor și mulțimea predecesorilor nodului.</p> <p>Numărul de vecini ai nodului i este egal cu cardinalul mulțimii de noduri adiacente nodului i.</p> <p>Arcul $[i,j]$ al grafului reprezintă nodul i și un nod j din lista de adiacență a nodului i din vectorul L $(L[0][j] \in L_i)$</p>

Din lista de adiacență **implementată static cu doi vectori** puteți obține următoarele informații:

Graf neorientat	Graf orientat
<p>Lungimea listei de adiacență a nodului i neizolat, cu eticheta mai mică decât n, este egală cu diferența dintre primul indice j ($j=i+1, n-1$) diferit de 0, din vectorul cap ($cap[j] \neq 0$), și indicele elementului de la care începe lista de adiacență a nodului i ($cap[j] - cap[i]$). Pentru nodul n, lungimea listei de adiacență este egală cu diferența dintre numărul total de elemente ale vectorului L, plus 1, și indicele elementului din care începe lista de adiacență a nodului n ($2 \times m + 1 - cap[n]$).</p> <p>Gradul unui nod i este egal cu lungimea listei de adiacență a nodului sau cu numărul de apariții ale etichetei nodului în vectorul listei L ($L[j]=i$, cu $j=1, 2 \cdot m$).</p> <p>Nodurile adiacente nodului i sunt nodurile a căror etichetă apare în lista de adiacență a nodului i din vectorul L.</p>	<p>Lungimea listei de adiacență a nodului i neizolat, cu eticheta mai mică decât n, se calculează la fel ca și în cazul grafului neorientat. Pentru nodul n, lungimea listei de adiacență este egală cu $(m+1 - cap[n])$.</p> <p>Gradul extern al nodului i este egal cu lungimea listei de adiacență a nodului.</p> <p>Gradul intern al nodului i este egal cu numărul de apariții ale etichetei nodului în vectoul listei L ($L[j]=i$, cu $j=1, m$).</p>
<p>Numărul de vecini ai nodului i este egal cu gradul nodului.</p> <p>Muchia $[i,j]$ a grafului reprezintă nodul i și un nod j din lista de adiacență a nodului i din vectorul L ($L[0][j] \in L_i$).</p>	<p>Succesori nodului i sunt nodurile a căror etichetă apare în lista de adiacență a nodului i din vectorul L.</p> <p>Predecesorii nodului i sunt nodurile j în a căror listă de adiacență din vectorul L apare nodul i.</p> <p>Nodurile adiacente nodului i sunt date de reunirea dintre mulțimea succesorilor și mulțimea predecesorilor nodului.</p>
<p>Numărul de vecini ai nodului i este egal cu gradul nodului.</p> <p>Muchia $[i,j]$ a grafului reprezintă nodul i și un nod j din lista de adiacență a nodului i din vectorul L ($L[0][j] \in L_i$).</p>	<p>Numărul de vecini ai nodului i este egal cu cardinalul mulțimii de noduri adiacente nodului i.</p> <p>Arcul $[i,j]$ al grafului reprezintă nodul i și un nod j din lista de adiacență a nodului i din vectorul L ($L[0][j] \in L_i$).</p>

Din lista de adiacență **implementată dinamic** puteți obține următoarele informații:

Graf neorientat	Graf orientat
<p>Numărul de elemente din toate listele simplu înălțuite este egal cu $2 \times m$ (dublul numărului de muchii).</p> <p>Lungimea listei de adiacență a nodului i este egală cu numărul de noduri ale listei ce are adresa nodului prim egală cu $L[i]$.</p>	<p>Numărul de elemente din toate listele simplu înălțuite este egal cu m (numărul de arce).</p> <p>Lungimea listei de adiacență a este egală cu numărul de noduri ale listei care are adresa nodului prim egală cu $L[i]$.</p>

Graf neorientat	Graf orientat
Gradul unui nod i este egal cu lungimea listei de adiacență.	Gradul extern al nodului i este egal cu lungimea listei de adiacență a nodului. Gradul intern al nodului i este egal, cu numărul de apariții ale etichetei nodului în toate listele simplu înălțuite.
Nodurile adiacente nodului i sunt nodurile a căror etichetă apare în lista ce are adresa nodului prim egală cu $L[i]$.	Succesorii nodului i sunt nodurile a căror etichetă apare în lista ce are adresa nodului prim egală cu $L[i]$. Predecesorii nodului i sunt nodurile j în ale căror liste (ce au adresa nodului prim egală cu $L[j]$) apare nodul i . Nodurile adiacente nodului i sunt date de reuniunea dintre mulțimea succesorilor și mulțimea predecesorilor nodului.
Numărul de vecini ai nodului i este egal cu gradul nodului.	Numărul de vecini ai nodului i este egal cu cardinalul mulțimii de noduri adiacente nodului i .
Muchia $[i,j]$ a grafului reprezintă nodul i și un nod j din lista ce are adresa nodului prim egală cu $L[i]$.	Arcul $[i,j]$ al grafului reprezintă nodul i și un nod j din lista ce are adresa nodului prim egală cu $L[i]$.

Temă

1. Scrieți lista de adiacență (folosind cele trei metode de implementare) a grafului neorientat G_4 . Folosind informațiile din lista de adiacență, determinați: gradul nodului 5, nodurile izolate și nodurile terminale.
2. Scrieți lista de adiacență a grafului orientat G_{14} . Se vor folosi cele trei metode de implementare. Ce proprietate are acest graf?
3. Scrieți lista de adiacență a grafului orientat G_8 (folosind cele trei metode de implementare). Folosind informațiile din lista de adiacență, determinați: gradul intern al nodului 5, gradul extern al nodului 4, succesorii și predecesorii nodului 2 și predecesorii nodului 3.
4. Scrieți lista de adiacență a grafului orientat G_{11} . Se vor folosi cele trei metode de implementare. Din lista de adiacență, determinați: gradul intern al nodului 5, gradul extern al nodului 2, nodurile adiacente nodului 5, succesorii și predecesorii nodului 4, nodurile terminale și nodurile izolate.
5. Scrieți lista de adiacență a grafului orientat G_{18} din figura 13. Se vor folosi cele trei metode de implementare.
6. Scrieți lista de adiacență a grafului G_{13} . Se vor folosi cele trei metode de implementare. Cum identificați – în fiecare implementare a listei de adiacență – nodul sursă al grafului?
7. Scrieți lista de adiacență a grafului G_{14} . Se vor folosi cele trei metode de implementare. Cum identificați – în fiecare implementare a listei de adiacență – nodul destinație al grafului?

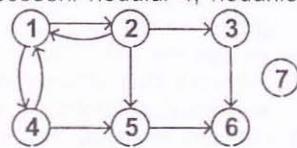


Fig 13

Implementarea algoritmilor pentru reprezentarea grafurilor cu lista de adiacență

1. Crearea listei de adiacență în **implementare statică** prin citirea listelor de vecini din fișier. Determinarea gradului unui nod. Obținerea matricei de adiacență din matricea listei de adiacență.

Într-un fișier text se găsesc următoarele informații despre un graf neorientat (și varianta orientată): pe prima linie, valorile pentru numărul de noduri n și numărul de muchii m , despărțite prin spațiu, iar pe următoarele n linii, despărțite prin spațiu, numărul de vecini ai unui nod și lista vecinilor (și varianta numărul de succesiuni ai unui nod și lista succesorilor).

Se citesc din fișierul text aceste informații și se creează lista de adiacență implementată cu matrice, respectiv cu vectori. Se afișează nodurile cu gradul cel mai mare (și varianta cu gradul extern cel mai mare). Se obține matricea de adiacență din lista de adiacență. Se salvează apoi matricea de adiacență într-un fișier text. În fișierul text se scriu: pe primul rând ordinul grafului, iar pe următoarele rânduri – linile matricei de adiacență. Funcția `citeste()` se folosește pentru a citi liste de adiacență din fișier, funcția `grad()` pentru a determina gradul (și varianta gradul extern) al unui nod, funcția `grad_max()` pentru a determina gradul maxim (și varianta gradul extern maxim) al nodurilor din graf, funcția `transpun()` pentru a obține matricea de adiacență din lista de adiacență, iar funcția `scrive()` pentru a scrie matricea de adiacență în fișier. Informațiile se citesc din fișierul text `graf13.txt`, pentru graful neorientat, și `graf14.txt`, pentru graful orientat, și se salvează în fișierul text `graf15.txt`, pentru graful neorientat, și `graf16.txt`, pentru graful orientat. Fișierele text `graf13.txt` și `graf14.txt` se creează cu un editor de texte. Pentru testarea programelor se folosesc graful neorientat G_1 și graful orientat G_9 .

Graful neorientat și graful orientat – implementarea cu matrice

```
#include<iostream.h>
int n,m,L[3][50],a[10][10];
//pentru graful neorientat
fstream f1("graf13.txt",ios::in),f2("graf15.txt",ios::out);
//pentru graful orientat
//fstream f1("graf14.txt",ios::in),f2("graf16.txt",ios::out);
void citeste()
{
    int i,j,x,k; f1>>n>>m;
    for(i=1;i<=n;i++) L[1][i]=i; j=n+1;
    for(i=1;i<=n;i++)
        {f1>>x;
         if (x==0) L[2][i]=0;
         else {L[2][i]=j;
                for (k=1;k<=x;k++) {f1>>L[1][j];
                                         if(k!=x) L[2][j]=j+1;
                                         else L[2][j]=0;
                                         j++;}
                }
        }
    f1.close();}
int grad(int i)
{
    int g,j=L[2][i];
    if (L[2][i]==0) g=0;
    else {g=1;
          while (L[2][j]!=0) {g++;j++;}
    }
    return g;}
int grad_max()
{
    int i,max=0;
    for (i=1; i<=n; i++) if (grad(i)>max) max=grad(i);
    return max;}
void transpun()
{
    int i,j;
    for (i=1; i<=n; i++)
        {j=L[2][i];
         if (L[2][i]!=0)
             {while (L[2][j]!=0) {a[i][L[1][j]]=1; j++;}
              a[i][L[1][j]]=1;})}
}
```

```

void scrie()
{int i,j; f2<<n<<endl;
 for (i=1; i<=n; i++)
 {for (j=1; j<=n; j++) f2<<a[i][j]<<" "; f2<<endl;}
f2.close();}
void main()
{int i; citeste();
 cout<<"Gradul cel mai mare este "<<grad_max()<<endl;
 cout<<"Nodurile cu gradul cel mai mare sunt: ";
 for (i=1; i<=n; i++) if (grad(i)==grad_max()) cout<<i<<" ";
transpne(); scrie();}

```

Graful neorientat și graful orientat – implementarea cu vectori.

```

#include<iostream.h>
int n,m, L[50],a[10][10],cap[10];
//pentru graful neorientat
fstream f1("graf13.txt",ios::in),f2("graf15.txt",ios::out);
//pentru graful orienat
//fstream f1("graf14.txt",ios::in),f2("graf16.txt",ios::out);
void citeste()
{int i,j=1,x,k; f1>>n>>m;
 for(i=1;i<=n;i++)
 {f1>>x;
 if (x==0) cap[i]=0;
 else {cap[i]=j;
 for (k=1;k<=x;k++) {f1>>L[j]; j++;}
}
 f1.close();}
int grad(int i)
{int g,j;
 if (cap[i]==0) g=0;
 else
 {if (i<n) {j=i+1;
 while (cap[j]==0 && j<=n) j++;
 if (j==n+1) g=2*m+1-cap[i];
 // g=m+1-cap[i]; pentru graful orientat
 else g=cap[j]-cap[i];}
 else g=2*m+1-cap[i];}
 return g;}
int grad_max() //este identică cu cea de la implementarea cu matrice
void transpne()
{int i,j;
 for (i=1; i<=n; i++)
 for (j=0;j<grad(i);j++) a[i][L[cap[i]+j]]=1;}
void scrie() //este identică cu cea de la implementarea cu matrice
void main() //este identică cu cea de la implementarea cu matrice}

```

2. Crearea listei de adiacență în implementare statică din matricea de adiacență. Salvarea listelor de adiacență într-un fișier.

Se citește din fișierul creat anterior (*graf15.txt*, respectiv *graf16.txt*) matricea de adiacență a grafului și se obține din ea lista de adiacență implementată cu matrice, respectiv cu vectori. Se salvează listele de adiacență într-un fișier text (*graf17.txt*, respectiv *graf18.txt*), astfel: pe prima linie, valorile pentru numărul de noduri *n* și numărul de muchii *m*, despărțite prin

spațiu, iar apoi, pe următoarele n linii, despărțite prin spațiu, numărul de vecini ai unui nod și lista vecinilor pentru graful neorientat, respectiv numărul de succesiuni ai unui nod și lista succesorilor pentru graful orientat. Funcția `citeste()` se folosește pentru a citi matricea de adiacență din fișier, funcția `transpune()` pentru a obține lista de adiacență din matricea de adiacență, iar funcția `scrie()` pentru a scrie listele de adiacență în fișier. Pentru testarea programelor se folosesc graful neorientat G_1 și graful orientat G_9 .

Graful neorientat și graful orientat – implementarea cu matrice

```
#include<iostream.h>
int n,m,L[3][50],a[10][10];
//pentru graful neorientat
fstream f1("graf15",ios::in),f2("graf17.txt",ios::out);
//pentru graful orientat
//fstream f1("graf16.txt",ios::in),f2("graf18.txt",ios::out);
void citeste()
{int i,j; f1>>n;
 for(i=1;i<=n;i++)
    for(j=1;j<=n;j++) {f1>>a[i][j]; if(a[i][j]==1) m++;} f1.close();
 m=m/2;} //numai pentru graful neorientat
void transpune()
{int i,j,k,g;
 for(i=1;i<=n;i++) L[1][i]=i; k=n+1;
 for(i=1;i<=n;i++)
 {for(j=1,g=0;j<=n;j++)
    if (a[i][j]==1) {L[1][k]=j; L[2][k]=k+1; k++; g++;}
    if (g==0) L[2][i]=0;
    else {L[2][i]=k-g; L[2][k-1]=0;}}
int grad(int i) {//identică cu cea de la implementarea cu matrice}
void scrie()
{int i,j,l,k; f2<<n<<" "<<m<<endl;
 for (i=1;i<=n;i++)
    if (L[2][i]==0) f2<<0<<endl;
    else {f2<<grad(i)<<" ";
           for (k=1;k<=grad(i);k++,j++) f2<<L[1][j]<<" ";
           f2<<endl;} f2.close();}
void main()
{citeste(); transpune(); scrie();}
```

Graful neorientat și graful orientat – implementarea cu vectori

```
#include<iostream.h>
int n,m,L[50],cap[10],a[10][10];
//pentru graful neorientat
fstream f1("graf13.txt",ios::in),f2("graf15.txt",ios::out);
//pentru graful orientat
//fstream f1("graf14.txt",ios::in),f2("graf16.txt",ios::out);
void citeste() {//este identică cu cea de la implementarea cu matrice}
void transpune()
{int i,j,k=1,g;
 for(i=1;i<=n;i++)
 {for(j=1,g=0;j<=n;j++) if (a[i][j]==1) {L[k]=j; k++; g++;}
    if(g==0) cap[i]=0; else cap[i]=k-g;}}
int grad(int i) {//identică cu cea de la implementarea cu vectori}
```

```

void scrie()
{int i,j; f2<<n<<" "<<m<<endl;
 for(i=1;i<=n;i++)
 {f2<<grad(i)<<" ";
 for(j=0;j<grad(i);j++) f2<<L[cap[i]+j]<<" ";
 f2<<endl;}
 f2.close();}
void main() { //este identică cu cea de la implementarea cu matrice}

```

3. Crearea liste de adiacență în **implementare dinamică** prin citirea listei muchiilor din fișier. Determinarea vecinilor și a gradului unui nod.

Într-un fișier text se găsesc următoarele informații despre un graf neorientat (și varianta orientată): pe prima linie, valorile pentru numărul de noduri n și numărul de muchii m , iar de pe următoarele m linii, câte o pereche de numere despărțite prin spațiu, care reprezintă etichetele nodurilor ce formează o muchie (arc). Se citesc din fișierul text aceste informații și se creează lista de adiacență implementată dinamic. Se afișează vecinii și gradul fiecărui nod. Funcția `init()` se folosește pentru a initializa cu valoarea **NULL** elementele vectorului `L` (pointerii nodului `prim` al listei de adiacență a fiecărui nod din graf), funcția `adauga_nod()` pentru a adăuga un nod înaintea nodului `prim` la lista simplu înlățuită a unui nod din graf, funcția `creare()` pentru a crea lista de adiacență, funcția `grad()` pentru a determina gradul (și varianta, gradul extern) al unui nod, funcția `afisare_vecini()` pentru a afișa vecinii fiecărui nod, iar funcția `afisare_grad()` pentru a afișa gradul fiecărui nod. Informațiile se citesc din fișierul text `graf11.txt` pentru graful neorientat și `graf12.txt` pentru graful orientat. Pentru testarea programelor se folosesc graful neorientat G_1 și graful orientat G_9 .

```

#include<iostream.h>
struct nod {int info;
            nod *urm;};
nod *L[20];
int n,m;
fstream f1("graf11.txt",ios::in);
void init()
{n>>m; for (int i=1;i<=n;i++) L[i]=NULL;}
void adauga_nod(nod *&prim, int y)
{nod *p=new nod; p->info=y; p->urm=prim; prim=p;}
void creare()
{int x,y;
 while (f>>x>>y) {adauga_nod(L[x],y);adauga_nod(L[y],x);}
 f.close();}
void afisare_vecini()
{for (int i=1;i<=n;i++)
 {cout<<"Vecinii nodului "<<i<<": ";
  for (nod *p=L[i];p!=NULL;p=p->urm) cout<<p->info<<" ";
  cout<<endl;}}
int grad(int i)
{int g=0;
 for (nod *p=L[i];p!=NULL; p=p->urm) g++; return g;}
void afisare_grade()
{for (int i=1;i<=n;i++)
 cout<<"Gradul nodului "<<i<<": "<<grad(i)<<endl;}
void main()
{init(); creare(); afisare_vecini(); afisare_grade();}

```



1. Scrieți un program care citește listele de adiacență ale grafului din fișierul text *graf13.txt*, pentru graful neorientat, respectiv din fișierul *graf14.txt*, pentru graful orientat, și care afișează nodurile izolate. Listele de adiacență se implementează dinamic și static, în cele două variante.
2. Scrieți un program care citește listele de adiacență ale grafului din fișierul text *graf13.txt*, pentru graful neorientat, respectiv din fișierul *graf14.txt*, pentru graful orientat, și care afișează muchiile (arcele) grafului. Listele de adiacență se implementează dinamic și static în cele două variante.
3. Scrieți un program care citește din două fișiere text, respectiv din *g5.txt* un graf orientat, reprezentat prin lista muchiilor, și din fișierul *g6.txt* un graf orientat, reprezentat prin lista vecinilor, și care verifică dacă cele două grafuri sunt identice (**Indicație**. Se reprezintă ambele grafuri prin matricea de adiacență și se compară cele două matrice de adiacență).

2.7.4.5. Aplicații practice

1. Într-un grup de n persoane s-au stabilit două tipuri de relații: de prietenie și de vecinătate. Scrieți un program care să citească matricele de adiacență ale celor două grafuri dintr-un fișier text (pe primul rând, numărul de persoane, și pe următoarele rânduri, în ordine, liniile fiecărei matrice de adiacență) și care să afișeze:
 - a. persoanele care sunt și prietene și vecini (**Indicație**. Se determină muchiile din graful intersecție a celor două grafuri neorientate);
 - b. cel mai mare număr de persoane care se găsesc într-un grup de vecini prieteni (**Indicație**. Se determină gradul maxim în graful intersecție a celor două grafuri.)
 Pentru testarea programului formați un grup cu 10 dintre colegii voștri și descrieți cu ajutorul a două grafuri cele două tipuri de relații.
2. Într-un grup de n persoane s-a stabilit o relație de cunoștință: persoana x este în relație cu persoana y dacă o cunoaște pe aceasta. Relația de cunoștință nu este reciprocă. O *celebritate* este o persoană care este cunoscută de toate persoanele din grup, dar care nu cunoaște nici o persoană (este un nod destinație al grafului). Un *necunoscut* este o persoană care cunoaște toate persoanele din grup dar nu este cunoscută de nici o persoană din grup (este un nod sursă al grafului). Un *singuratici* este o persoană care cunoaște o singură persoană din grup sau este cunoscută de o singură persoană din grup (este un nod terminal al grafului). Un *străin de grup* este o persoană care nu cunoaște nici o persoană din grup și nu este cunoscută de nici o persoană din grup (este un nod izolat al grafului). Demonstrați că în grup nu poate exista decât o singură *celebritate* și un singur *necunoscut*. Scrieți un program care să citească dintr-un fișier matricea de adiacență a grafului și care să afișeze:
 - a. dacă există o *celebritate*, să se precizeze persoana, iar dacă nu există, să se precizeze: persoana care cunoaște cele mai puține persoane și persoana care este cunoscută de cele mai multe persoane;
 - b. dacă există un *necunoscut*, să se precizeze persoana, iar dacă nu există, să se precizeze: persoana care este cunoscută de cele mai puține persoane și persoana care cunoaște cele mai multe persoane;
 - c. dacă există *singuratici*, să se precizeze persoanele;
 - d. dacă există *străini de grup*, să se precizeze persoanele;
 - e. câte persoane cunosc doar două persoane din grup și sunt cunoscute la rândul lor de trei persoane din grup (nodurile care au gradul intern egal cu 3, iar gradul extern egal cu 2);

- f. câte persoane sunt cunoscute de un număr de membri egal cu numărul de membri pe care îi cunoșc (nodurile care au gradul intern egal cu gradul extern);
- g. care sunt persoanele care se cunosc reciproc (perechile de noduri între care există arce duble).
3. La graful județelor, adăugați un nod cu eticheta 0, care reprezintă exteriorul țării, și muchiile corespunzătoare, care evidențiază județele limitrofe. Creați cu un editor de texte fișierul *judete.txt* care să conțină următoarele informații: pe primul rând, numărul de județe, pe următoarele rânduri matricea de adiacență, pe următorul rând, în ordinea etichetelor nodurilor, denumirile județelor separate prin spațiu (pentru județele care au un nume format din mai multe cuvinte folosiți ca separator linia de subliniere), pe următorul rând numele vecinilor României separate prin spațiu, și apoi, câte un rând pentru fiecare județ limitrof, în care scrieți separate prin spațiu următoarele informații: eticheta nodului și țara (țările) cu care se învecinează, precizate prin numărul de ordine din lista numelor. Scrieți un program care să citească aceste informații din fișier și să le transpună într-o implementare a grafului, adecvată problemei, și care să afișeze:
- județele care au cele mai multe județe vecine – se va preciza numărul maxim de vecini și numele județelor care au această proprietate;
 - județele care au cele mai puține județe vecine – se va preciza numărul minim de vecini și numele județelor care au această proprietate;
 - județele de la graniță – pentru fiecare județ se va preciza numele său și numele țărilor cu care se învecinează.
4. Într-o zonă turistică există n localități. Între unele localități există legături directe, fie prin șosele naționale, fie prin șosele județene. Legăturile directe sunt caracterizate de lungimea drumului, măsurată în kilometri. Se vor folosi două grafuri: $G_n = (X, U_n)$ pentru legăturile prin șosele naționale și $G_j = (X, U_j)$ pentru legăturile prin șosele județene. Cele două grafuri se vor citi din două fișiere text, care conțin pentru fiecare legătură directă, pe câte un rând, separate prin spațiu, cele două etichete ale nodurilor asociate localităților și distanța dintre localități. Scrieți un program care să citească aceste informații din fișier și să le transpună într-o implementare a grafului, adecvată problemei, și care să afișeze:
- localitățile la care nu ajung drumuri naționale (nodurile izolate, în primul graf);
 - cea mai scurtă legătură directă dintre două localități – se va preciza eticheta nodurilor asociate localităților, distanța dintre localități și tipul șoselei prin care se asigură legătura;
 - pentru două localități precizate, a și b (etichetele nodurilor a și b se citesc de la tastatură), să se precizeze dacă există legătură directă; dacă există, să se mai precizeze tipul șoselei și distanța dintre localități;
 - pentru o localitate p (eticheta p a nodului se citește de la tastatură), să se afișeze toate localitățile cu care are legături directe și distanța până la aceste localități (**Indicație**. Se determină graful reuniune a celor două grafuri.);
 - localitatea care are cele mai multe legături directe cu alte localități – și să se afișeze localitățile cu care are legături (nodul cu gradul maxim în graful reuniune).
5. Într-un munte există n grote. Fiecare grotă i se găsește la înălțimea h_i față de baza muntelui. Între unele grote există comunicare directă, prin intermediul unor tuneluri. Există și grote care comunică cu exteriorul. Desenați o rețea ipotetică de grote și construiți matricea de adiacență a grafului neorientat asociat (grotele sunt nodurile, iar

tunelurile sunt muchiile). Înălțimile grotelor se vor memora într-un vector. Găsiți o modalitate de a evidenția grotete care comunică cu exteriorul. (Indicație. Adăugați la graf nodul 0, care reprezintă exteriorul muntelui.) Scrieți matricea de adiacență și vectorul cu înălțimile grotelor, în fișierul text *grote.txt*. Scrieți un program care să citească matricea de adiacență și vectorul din fișier – și care să afișeze:

- a. numărul de tuneluri de comunicare (numărul de muchii ale grafului);
 - b. grotete care au legătură cu exteriorul (nodurile i pentru care există muchie cu nodul 0);
 - c. grotete care comunică direct cu cele mai multe grote (nodurile cu cel mai mare grad);
 - d. grotete care nu comunică prin tuneluri cu alte grote (nodurile izolate);
 - e. cea mai înaltă grotă și cea mai joasă grotă care comunică cu exteriorul;
 - f. pentru o grotă a cărei etichetă se citește de la tastatură, să se afișeze grotete cu care comunică direct, precizând pentru fiecare tunel dacă suie, coboară sau este la același nivel cu grota.
6. Rețeaua de străzi dintr-un oraș este formată din străzi cu două sensuri și străzi cu sens unic de circulație. Ea poate fi reprezentată printr-un graf orientat, în care intersecțiile sunt nodurile, iar arcele – sensul de circulație pe străzile care leagă două intersecții (traficul auto). Nodurile sunt intersecții de cel puțin 3 străzi. Pentru a stabili prioritatea în intersecții și pentru a fluidiza traficul intersecțiile vor fi modernizate. Pentru modernizare se vor folosi panouri cu semne de circulație, semafoare – și se vor amenaja sensuri giratorii. Pentru fiecare stradă din care se poate intra în intersecție, se montează în intersecție un panou cu semn pentru prioritate. Pentru fiecare stradă pe care nu se poate intra din intersecție se montează în intersecție un panou cu semnul de interzicere a circulației. În toate intersecțiile vor fi montate panouri cu semne de circulație, corespunzător străzilor incidente cu intersecția. Fiecare dintre aceste mijloace de modernizare are un cost: panoul cu semn de circulație – costul **c1**, semaforul – costul **c2**, și sensul giratoriu – costul **c3**. Pentru a stabili modul în care este modernizată fiecare intersecție, intersecțiile au fost clasificate în intersecții mici (intersecții cu 3 străzi, în care vor fi montate numai panouri cu semne de circulație), intersecții mari (intersecții cu 4 străzi, care vor fi semaforizate) și intersecții foarte mari (intersecții cu peste 4 străzi, în care se vor amenaja sensuri giratorii). Desenati o rețea ipotetică de străzi și construji matricea de adiacență a grafului orientat asociat. Scrieți matricea de adiacență în fișierul text *strazi.txt*. Scrieți un program care să citească matricea de adiacență din fișier și care să afișeze:
 - a. intersecțiile la care nu se poate ajunge din nici o altă intersecție (nodurile care nu au predecesori);
 - b. intersecțiile de la care nu se poate ajunge la o nici o altă intersecție (nodurile care nu au succesori);
 - c. dacă există intersecții fără nici o intersecție succesor sau fără nici o intersecție predecesor, să se corecteze desenul rețelei (prin adăugarea unui număr minim de arce), astfel încât să nu existe asemenea intersecții – și să se actualizeze și fișierul *strazi.txt*.
 - d. intersecțiile la care se poate ajunge direct din cele mai multe intersecții (nodurile care au cel mai mare grad intern);
 - e. intersecțiile de la care se poate ajunge direct la cele mai multe intersecții (nodurile care au cel mai mare grad extern);
 - f. numărul de străzi pe care se circulă în ambele sensuri (numărul de perechi de noduri i și j pentru care, în matricea de adiacență, elementele $a[i][j]$ și $a[j][i]$ sunt egale cu 1);

- g. numărul de intersecții mici, mari și foarte mari (clasificarea nodurilor în funcție de numărul de noduri adiacente: noduri cu 3 noduri adiacente, cu 4 noduri adiacente și cu cel puțin 5 noduri adiacente);
- h. numărul de panouri cu semne pentru prioritate care se vor folosi (suma gradelor interne ale nodurilor);
- i. numărul de panouri cu semne pentru interzicerea circulației care se vor folosi (diferența dintre suma gradelor externe ale nodurilor și numărul de străzi cu sens dublu de circulație);
- j. numărul de semafoare care se vor monta (suma gradelor interne ale nodurilor care au 4 noduri adiacente);
- k. costul de modernizare necesar pentru fiecare intersecție și costul total al modernizării.
7. Pe un munte există mai multe parcele cu fânețe ale sătenilor. Unele fânețe au acces direct la drumul sătesc, altele nu. Între proprietarii parcelelor vecine există relație de prietenie – sau nu. Dacă doi proprietari vecini sunt prieteni, își permit unul altuia accesul pe propria parcelă. Pentru a transporta fânul, sătenii care nu au acces la drumul sătesc, trebuie să treacă peste parcelele altor săteni, ca să ajungă la el. Un proprietar este considerat izolat dacă nu are acces direct la drumul sătesc și nici nu este în relație de prietenie cu vreunul dintre vecinii lui. Se vor folosi două grafuri: unul pentru a reprezenta relația de vecinătate a fânețelor, iar altul pentru a reprezenta relația de prietenie dintre proprietari. Desenați o hartă ipotetică a fânețelor și stabiliți relații ipotetice de prietenie între vecini. Construiți matricele de adiacență ale celor două grafuri asociate. Scrieți matricele de adiacență în fișierele text *fanele.txt* și *prietenii.txt*. Scrieți un program care să citească matricele de adiacență din fișiere și care să furnizeze următoarele informații:
- dacă există proprietari izolați, să se afișeze lista proprietarilor vecini cu care trebuie să stabilească relații de prietenie, pentru a ajunge la drumul sătesc, și să se identifice vecinii care au acces direct la drumul sătesc;
 - care este proprietarul cel mai neprietenos (care are cel mai mare procent de vecini cu care nu este prieten).
8. Se analizează migrația unei specii de păsări călătoare pe perioada unui an. Migrația are două etape: migrația de toamnă, când păsările migrează din zonele reci către zonele calde (migrația de la rece la cald), și migrația de primăvară, când păsările migrează din zonele calde către zonele reci (migrația de la cald la rece). Fiecare etapă a migrației va fi reprezentată printr-un graf orientat. În graful migrației de la rece la cald, nodurile care aparțin zonei reci au numai succesi, iar nodurile care aparțin zonei calde au numai predecesori. În graful migrației de la cald la rece nodurile care aparțin zonei calde au numai succesi, iar nodurile care aparțin zonei reci au numai predecesori. Fiecărui nod i se asociază coordonatele geografice. Desenați câte o hartă ipotetică pentru fiecare etapă de migrație. Construiți, cu un editor de texte, fișierul *migratie1.txt*, care va conține pe prima linie **n1**, numărul de noduri ale primului graf, pe următoarele **n1** linii, matricea de adiacență a grafului, și apoi, pe următoarele **n1** linii, coordonatele geografice ale fiecărui nod de pe hartă (sunt 6 entități de informație, separate prin spațiu: două valori numerice întregi pentru latitudine, în grade și minute, și un caracter pentru emisferă, două valori numerice întregi pentru longitudine, în grade și minute, și un caracter pentru meridian). Construiți cu un editor de texte fișierul *migratie2.txt*, care va conține același tip de informații, pentru cel de al doilea graf al migrației. Scrieți un program care să

citească aceste informații din fișier, să le transpună într-o implementare a grafului adecvată problemei (**recomandare** – implementare prin matrice de adiacență și vector de structuri pentru coordonatele geografice) și care să afișeze următoarele informații:

- dacă cele două grafuri sunt corecte (dacă $a[i][j]=1$, atunci $a[j][i]=0$ – și produsul dintre gradul intern și gradul extern ale fiecărui nod trebuie să fie 0; în plus, trebuie ca nodurile din zona caldă să coincidă – în ambele grafuri);
- dacă în graful migrației de la rece la cald există un nod destinație al grafului – și ce semnificație are existența lui pentru migrație;
- dacă păsările s-au reîntors în aceleași zone, primăvara (nodurile din zona rece coincid în cele două grafuri); în caz contrar, să se specifice noile locații apărute în zona rece în care a ajuns specia respectivă;
- să se obțină matricea de adiacență a hărții migrației, pe întreaga perioadă a anului, prin reuniunea celor două grafuri (dacă după migrația de primăvară au apărut noi locații pe hartă în zona rece, se vor adăuga la primul graf ca noduri izolate).

2.7.5. Grafuri speciale

Se definesc următoarele grafuri speciale:

- graful nul;
- graful complet.

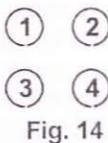
2.7.5.1. Graful nul

Graful $G=(X,U)$ se numește **graf nul** dacă mulțimea U este vidă ($U=\emptyset$), adică graful nu are muchii.

Reprezentarea sa în plan se face prin noduri izolate.

Exemplu:

Graful $N=(X,V)$ – unde mulțimea $X=\{1,2,3,4\}$ este mulțimea nodurilor, iar mulțimea $V=\emptyset$ este mulțimea muchiilor – este un graf nul (graful are noduri dar nu are muchii) – figura 14.



Observație. Matricea de adiacență a unui graf nul este matricea zero (nu conține nici un element cu valoarea 1).

Fig. 14

2.7.5.2. Graful complet

Un graf cu n noduri este un **graf complet** dacă are proprietatea că, oricare ar fi două noduri ale grafului, ele sunt adiacente. El se notează cu K_n .

Observații.

- Se poate construi un singur graf neorientat complet, cu n noduri, deoarece între două noduri, x și y , există o singură muchie $[x,y]$. În figura 15 este prezentat graful neorientat complet K_4 . El are 6 muchii. Desenați grafurile neorientate complete K_5 și K_6 . Numărați câte muchii au aceste grafuri.

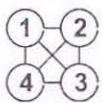


Fig. 15

Teorema 6

Numărul m de muchii ale unui graf neorientat complet, cu n noduri (K_n), este:

$$m = n \times \frac{n - 1}{2}$$

Demonstrație. Numărul de muchii este dat de numărul de submulțimi de 2 elemente care se pot forma dintr-o mulțime cu n elemente, adică $m = C_n^2$.

2. Se pot construi **mai multe grafuri orientate complete**, cu n noduri, deoarece două noduri x și y pot fi adiacente în trei situații: există arcul $[x,y]$, există arcul $[y,x]$ sau există arcele $[x,y]$ și $[y,x]$.

Teorema 7

Numărul de grafuri orientate complete care se pot construi cu n noduri este egal cu

$$n_k = 3^{C_n^2}$$

Demonstrație. Deoarece numărul de submulțimi de 2 noduri care se pot forma dintr-o mulțime de n noduri este $a = C_n^2$, și pentru fiecare pereche astfel definită se pot defini trei situații de adiacență, rezultă că numărul de grafuri complete care se pot construi este de 3^a .

Exemplu. Pentru $n=4$ se pot defini $3^6=729$ grafuri orientate complete. În figura 16 sunt prezentate patru dintre acestea. Definiți alte patru grafuri complete cu 4 noduri.

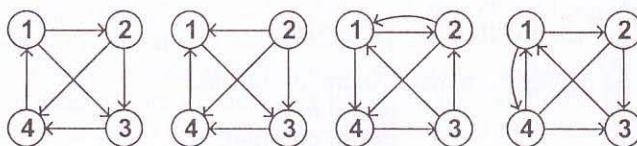


Fig. 16

Observații

- În cazul matricei de adiacență a unui graf neorientat complet, valoarea fiecărui element care nu se găsește pe diagonala principală este 1.
- În cazul matricei de adiacență a unui graf orientat complet – pentru orice pereche de noduri, i și j , diferite între ele ($i \neq j$) – $a[i][j]=1$ sau $a[j][i]=1$.
- Numărul minim de arce** într-un graf orientat complet cu n noduri este egal cu numărul de muchii ale grafului neorientat complet K_n .
- Numărul maxim de arce** într-un graf orientat complet cu n noduri este egal cu dublul numărului de muchii ale grafului neorientat complet K_n .

Algoritmi pentru prelucrarea grafurilor complete

1. **Algoritm pentru a determina numărul minim de arce care trebuie adăugate la un graf orientat, pentru a obține un graf orientat complet.**

Algoritmul. Se numără perechile de noduri i și j ($i \neq j$) între care nu există nici un arc.

Implementarea algoritmului În program, informațiile despre graful orientat se citesc din fișierul text *gc.txt*: de pe prima linie numărul de noduri, și apoi, de pe următoarele rânduri matricea de adiacență.

```
#include<iostream.h>
int n,a[10][10];
fstream f("gc.txt",ios::in);
void citeste() { //se citește matricea de adiacență din fisier}
void main()
{int i,j,m=0; citeste();
for(i=2;i<=n;i++)
    for(j=1;j<i;j++)
        if(a[i][j]==0 && a[j][i]==0) m++;
cout<<"Numarul de arce care trebuie adaugate este "<<m; }
```

2. Algoritm pentru a determina numărul maxim de noduri izolate pe care poate să le conțină un graf neorientat care are n noduri și m muchii.

Algoritmul. Se identifică graful complet care se poate forma astfel încât să consume cât mai multe muchii (m_{max}) din cele m muchii ale grafului ($m_{max} \leq m$). Graful complet astfel obținut are n_1 noduri:

$$m_{max} = n_1 \times \frac{n_1 - 1}{2} \leq m$$

Numărul de noduri n_1 este partea întreagă din rădăcina pozitivă a ecuației de gradul II:

$$n_1 = \left\lceil \frac{1 + \sqrt{1 + 8 \times m}}{2} \right\rceil$$

Pentru diferența de muchii rămase ($m - m_{max}$) mai este necesar un nod, pentru a lega aceste muchii de nodurile grafului complet care s-a format. Numărul de noduri izolate este: $n - n_1 - 1$.

```
#include<iostream.h>
#include<math.h>
void main()
{int m,n,n1; cout<<"muchii= "; cin>>m; cout<<"noduri= "; cin>>n;
n1=(int)((1+sqrt(1+8*m))/2);
cout<<"Numarul maxim de noduri izolate este "<<n-n1-1;}
```

Temă

1. Scrieți un program care citește, din fișierul text *graf_c1.txt*, informații despre un graf neorientat (de pe prima linie, numărul de noduri, apoi matricea de adiacență), și care verifică dacă este un graf complet.
2. Scrieți un program care citește, din fișierul *graf_c2.txt*, informații despre un graf orientat (de pe prima linie, numărul de noduri, apoi matricea de adiacență) – și care verifică dacă este un graf complet.

2.7.6. Grafuri deriveate dintr-un graf

Se definesc următoarele grafuri:

- graful parțial;
- subgraful;
- graful complementar.

2.7.6.1. Graful parțial

Fie graful $G = (X, U)$ și mulțimea $V \subseteq U$. Graful $G_p = (X, V)$ se numește **graf parțial** al grafului G .

Altfel spus, un graf parțial al grafului G este el însuși sau un graf care s-a obținut prin eliminarea unor muchii (arce) din graful G .

Exemple:

1. Pentru graful neorientat $G_1 = (X_1, U_1)$, definit anterior, graful $G_{1p} = (X_1, U_{1p})$, definit astfel:
 - mulțimea nodurilor este $X_1 = \{1, 2, 3, 4, 5, 6, 7, 8\}$.
 - mulțimea muchiilor este $U_{1p} = \{[1, 2], [1, 3], [1, 4], [2, 3], [2, 5], [3, 4], [6, 7]\}$.
 este subgraf al grafului G_1 – figura 17.

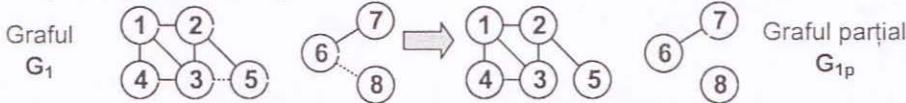
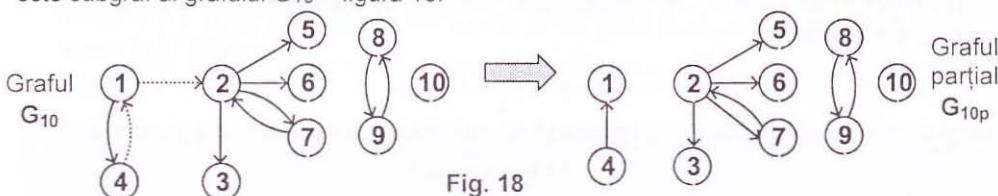


Fig. 17

G10p

2. Pentru graful orientat $G_{10} = (X_{10}, U_{10})$ definit anterior, graful $G_{10p} = (X_{10}, U_{10p})$ definit astfel:
 → mulțimea nodurilor este $X_{10} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.
 → mulțimea arcelor este $U_{10p} = \{[2, 3], [2, 5], [2, 6], [2, 7], [4, 1], [7, 2], [8, 9], [9, 8]\}$.
 este subgraf al grafului G_{10} – figura 18.



Teorema 8

Numărul de grafuri parțiale ale unui graf cu m muchii (arce) este egal cu 2^m .

Demonstrație. Grafurile parțiale se pot obține: fără eliminarea unei muchii (arc) – obținându-se graful inițial; prin eliminarea unei muchii (unui arc) – obținându-se grafurile parțiale cu $m-1$ muchii; ...; prin eliminarea a $m-1$ muchii (arce) – obținându-se grafurile parțiale cu o muchie (un arc); și a tuturor muchiilor (arcelor) – obținându-se un graf parțial numai cu noduri și fără muchii (arce), adică grafa vid:

Numărul de grafuri parțiale care se pot obține cu m muchii (arce):

$$C_m^m$$

Numărul de grafuri parțiale care se pot obține cu $m-1$ muchii (arce):

$$C_m^{m-1}$$

Numărul de grafuri parțiale care se pot obține cu $m-2$ muchii (arce):

$$C_m^{m-2}$$

.....

Numărul de grafuri parțiale care se pot obține cu o muchie (un arc):

$$C_m^1$$

Numărul de grafuri parțiale care se pot obține fără nici o muchie (nici un arc):

$$C_m^0$$

Numărul total de grafuri parțiale este: $C_m^m + C_m^{m-1} + C_m^{m-2} + \dots + C_m^1 + C_m^0 = 2^m$.

Algoritmi pentru prelucrarea grafurilor parțiale

1. Generarea tuturor grafurilor parțiale ale unui graf neorientat.

Algoritm. Se folosește metoda **backtracking**. În stivă se vor genera muchiile grafului parțial. Deoarece graful parțial poate avea p muchii ($0 \leq p \leq m$), se va apela repetat subprogramul **bț()**, la fiecare apel generându-se variantele cu p muchii. Fiecare apel al subprogramului **bț()** trebuie să genereze C_n^p de muchii (arce) din graf. La fiecare apel al subprogramului, soluția se obține atunci când în stivă s-au generat cele p muchii ale grafului parțial. Evidența muchiilor este păstrată în **matricea de incidentă**.

Implementarea algoritmului. Se citește din fișierul text *graf15.txt* matricea de adiacență a unui graf neorientat. Pentru fiecare graf parțial se afișează muchiile. Matricea de incidentă **b** se obține din matricea de adiacență cu funcția **transformare()**. În variabila **nr** se contorizează numărul de grafuri parțiale generate. În funcția **tipar()** se afișează graful parțial generat. Pentru testarea programului se folosește graful G_1 .

```
#include<iostream.h>
fstream f("graf13.txt", ios::in);
typedef int stiva[100];
int n,m,p,k,ev,as,a[10][10],b[10][20],nr;
stiva st;
void citeste()
```

```

{int i,j; f1>>n>>x;
for (i=1;i<=n;i++)
    for (j=1;j<=n;j++) {f1>>a[i][j]; if (a[i][j]==1) m++;}
m=m/2; f1.close();}
void transformare()
{int i,j,k=1;
for (i=1;i<=n;i++)
    for (j=1;j<=i;j++) if (a[i][j]==1) {b[i][k]=1; b[j][k]=1; k++;}
void init() {st[k]=0;}
int succesor()
{if (st[k]<m) {st[k]=st[k]+1; return 1;} else return 0;}
int valid()
{if (k>1 && st[k]<st[k-1]) return 0;
for (int i=1;i<k;i++) if (st[k]==st[i]) return 0;
return 1;}
int solutie() {return k==p;}
void tipar()
{int i,j; nr++;
cout<<"Graful parțial "<<nr<<endl; cout<<"Muchiile: ";
for(i=1;i<=p;i++)
    {for(j=1;j<=n;j++) if (b[j][st[i]]==1) cout<<j<<" ";
    cout<<" ; ";}
cout<<endl;}
void bt() {//partea fixă a algoritmului backtracking}
void main()
{citere(); transformare();
for(p=m;p>=0;p--) bt();}

```

2. Verificarea dacă un graf G_p este graf parțial al unui graf G .

Algoritmul. Se verifică dacă cele două grafuri au același număr de noduri și dacă graful G_p nu conține muchii care nu există în graful G .

Implementarea algoritmului. Se citesc, din două fișiere text *g1p.txt* și *g2p.txt*, informații despre două grafuri neorientate (sau orientate): de pe prima linie, numărul de noduri, și apoi, de pe următoarele rânduri, matricea de adiacență. Matricele de adiacență ale celor două grafuri sunt *a1* și *a2*, cu dimensiunea *n*, respectiv *m*. Funcția *grafp()* verifică dacă graful G_p este graf parțial al grafului G .

```

#include<iostream.h>
fstream f1("g1p.txt",ios::in),f2("g2p.txt",ios::in);
int m,n,a1[10][10],a2[10][10];
int grafp()
{if (m!=n) return 0;
else for (int i=1;i<=n;i++)
    for(int j=1;j<=n;j++) if (a2[i][j]==1 && a1[i][j]==0) return 0;
return 1;}
void main()
{int i,j; f1>>n;
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++) f1>>a1[i][j]; f1.close();
f2>>m;
for(i=1;i<=m;i++)
    for(j=1;j<=m;j++) f2>>a2[i][j]; f2.close();
}

```

```
if(grafp()) cout<<"este graf parțial ";
else cout<<"nu este graf parțial";}
```

3. Obținerea unui graf parțial G_p al unui graf G .

Algoritmul. Se elimină din graful G muchiile, în funcție de condiția impusă de problemă. Reprezentarea cea mai adekvată pentru graf este matricea de adiacență, în care se atribuie valoarea 0 elementelor $a[i][j]$ și $a[j][i]$, corespunzătoare muchiilor $[i,j]$ care trebuie eliminate..

Implementarea algoritmului. Se citesc, din fișierul text *g3p.txt*, informații despre graful neorientat: de pe prima linie numărul de noduri n și eticheta unui nod x , și apoi, de pe următoarele rânduri, matricea de adiacență a grafului. Informațiile despre graful parțial obținut se scriu în fișierul text *g4p.txt*, astfel: pe primul rând numărul de noduri n și numărul de muchii m și pe următoarele m rânduri – muchiile, sub formă de perechi de etichete de noduri despărțite prin spațiu. Cerința este să se obțină subgraful prin eliminarea tuturor muchiilor care au la extremități un nod cu grad par și nodul x . În vectorul v se memorează nodurile care au grad par. Funcția *citește()* se folosește pentru a citi informațiile despre matricea de adiacență a grafului din fișierul text, funcția *scrive()* pentru a scrie în fișierul text informațiile despre lista muchiilor grafului parțial, funcția *grad()* pentru a determina gradul unui nod, iar funcția *graf_partial()* pentru a obține graful parțial. Pentru a obține graful parțial, se caută toate muchiile care au la extremități un nod cu gradul par și nodul x (muchiile $[v[i],x]$ pentru care $a[v[i]][x]==1$) și se elimină aceste muchii. Pentru testarea programului se va folosi graful G_3 și nodul 6.

```
#include<iostream.h>
fstream f1("g3p.txt",ios::in),f2("g4p.txt",ios::out);
int a[20][20],n,m,x,v[10];
void citește() { //se citește matricea de adiacență din fișier}
int grad(int i)
{int j,g=0;
 for (j=1;j<=n;j++) g+=a[i][j]; return g;}
void graf_partial()
{int i,k=0;
 for (i=1;i<=n;i++)
 if (grad(i)%2==0) {k++; v[k]=i;}
 for (i=1;i<=k;i++)
 if (a[v[i]][x]==1) {a[v[i]][x]=0; a[x][v[i]]=0; m--;}
void scrive()
{int i,j; f2<<n<<" "<<m<<endl;
 for (i=1;i<=n;i++)
 for (j=1;j<i;j++) if (a[i][j]==1) f2<<i<<" "<<j<<endl;
 f2.close();}
void main() {citește(); graf_partial(); scrive();}
```

Temă



1. Scrieți un program care citește, din fișierul text *graf16.txt*, matricea de adiacență a grafului orientat G_9 și care generează toate **grafurile parțiale** care se pot obține. Informațiile despre grafurile parțiale se vor salva în fișierul text *graf_p16.txt*. Pentru fiecare graf parțial generat se va scrie pe un rând textul *Graful parțial*, urmat de numărul de ordine al grafului generat, iar pe rândul următor textul *Arcele* urmat de lista de arce a grafului generat.
2. Scrieți un program care citește, din două fișiere text *graf_1p.txt* și *graf_2p.txt* informații despre matricele de adiacență a două grafuri neorientate (și varianta grafuri orientate):

de pe prima linie numărul de noduri, apoi matricea de adiacență, și care verifică dacă unul dintre grafuri este graf parțial al celuilalt. În caz afirmativ, se afișează care este graful și care este graful parțial. (Indicație. Dacă cele două grafuri au același ordin n , se identifică graful care are mai multe muchii (m) și numărul de muchii ale celuilalt graf – p ($m \geq p$). Pentru graful cu m muchii, se generează toate matricele de adiacență ale grafurilor parțiale cu m muchii și n noduri – și se verifică dacă una dintre ele este identică cu matricea de adiacență a celuilalt graf.)

3. Scrieți un program care citește din fișierul text *graf19.txt* matricea de adiacență a unui graf neorientat și care generează un graf parțial prin eliminarea muchiilor care au la extremități un nod ce are gradul minim și un nod ce are gradul maxim în graf. Informațiile despre graful parțial obținut se scriu într-un fișier text sub forma listei de muchii. Pentru testarea programului se va folosi graful G_4 .

2.7.6.2. Subgraful

Fie graful $G = (X, U)$. Graful $G_s = (Y, V)$ se numește **subgraf** al grafului G dacă $Y \subseteq X$ (subgraful conține numai noduri ale grafului) și muchiile (arcele) din mulțimea V sunt toate muchiile (arcele) din mulțimea U care au ambele extremități în mulțimea de noduri Y . Se spune că subgraful G_s este **indus** sau **generat** de mulțimea de noduri Y .

Altfel spus, un subgraf al grafului G este el însuși sau un graf care s-a obținut prin suprimarea din graful G a unor noduri și a tuturor muchiilor (arcelor) incidente cu aceste noduri.

Exemple:

1. Pentru graful neorientat $G_1 = (X_1, U_1)$, definit anterior, graful $G_{1s} = (Y_1, V_1)$, definit astfel:
 \rightarrow mulțimea nodurilor este $Y_1 = \{1, 2, 4, 5, 6, 7\}$.
 \rightarrow mulțimea muchiilor este $V_1 = \{\{1, 2\}, \{1, 4\}, \{2, 5\}, \{6, 7\}\}$.

este subgraf al grafului G_1 – figura 19.

G_{1s}

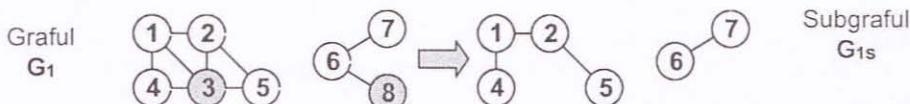


Fig. 19

2. Pentru graful orientat $G_{10} = (X_{10}, U_{10})$, definit anterior, graful $G_{10s} = (Y_{10}, V_{10})$ definit astfel:
 \rightarrow mulțimea nodurilor este $Y_{10} = \{1, 2, 3, 6, 7, 9, 10\}$.
 \rightarrow mulțimea arcelor este $V_{10} = \{\{1, 2\}, \{2, 1\}, \{2, 3\}, \{2, 6\}, \{2, 7\}, \{7, 2\}\}$.
este subgraf al grafului G_{10} – figura 20.

G_{10s}

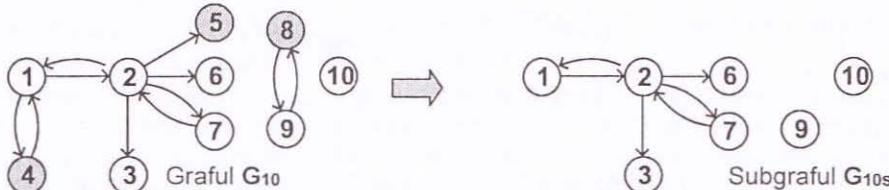


Fig. 20

Teorema 9

Numărul de subgrafuri ale unui graf cu n noduri este egal cu $2^n - 1$.

Demonstrație. Subgrafurile se pot obține: prin eliminarea niciunui nod (obținându-se graful inițial); a unui nod (obținându-se subgrafurile cu $n-1$ noduri); ...; a $n-1$ noduri (obținându-se subgrafurile cu un nod);

Numărul de subgrafuri care se pot obține cu n noduri:

C_n^R

Numărul de subgrafuri care se pot obține cu $n-1$ noduri:

$$\mathbb{C}^{n-1}$$

Numărul de subgrafuri care se pot obține cu $n-2$ noduri:

$$C^{n-2}$$

Numărul de subgrafuri care se pot obține cu 1 nod:

C1

Numărul total de subgrafuri este: $C_n^n + C_n^{n-1} + C_n^{n-2} + \dots + C_n^1 = 2^n - 1$.

Algoritmi pentru prelucrarea subgrafurilor

1. Generarea tuturor subgrafurilor unui graf.

Algoritmul. Se folosește metoda backtracking. În stivă se generează nodurile subgrafului. Deoarece subgraful poate avea p noduri ($1 \leq p \leq n$), se va apela repetat subprogramul `bt()`, la fiecare apel generându-se variantele cu p noduri. Fiecare apel al subprogramului `bt()` trebuie să genereze C_n^p de noduri din graf. La fiecare apel al subprogramului, soluția se obține atunci când în stivă s-au generat cele p noduri ale subgrafului. Pentru nodurile generate în stivă se afisează muchiile (arcele) care există în graf.

Implementarea algoritmului. Se citesc, din fișierul text *graf1.txt*, matricea de adiacență a unui graf neorientat, respectiv, din fișierul *graf2.txt*, matricea de adiacență a unui graf orientat. Pentru fiecare subgraf generat, se afișează numărul de noduri și muchiile (arcele). În variabila *nr* se contorizează numărul de subgrafuri generate. În funcția *tipar()* se afișează subgraful generat. Pentru testarea programelor se folosesc graful neorientat G_1 și graful orientat G_9 .

```

#include<fstream.h>
fstream f("graf1.txt",ios::in);
// fstream f("graf2.txt",ios::in); pentru graful orientat
typedef int stiva[100];
int n,p,k,ev,as,a[10][10],nr;
stiva st;
void citeste() {//se citește matricea de adiacență din fișier}
void init() {st[k]=0;}
int succesor()
{if (st[k]<n) {st[k]=st[k]+1; return 1;} else return 0;}
int valid()
{if (k>1 && st[k]<st[k-1]) return 0;
 for (int i=1;i<k;i++) if (st[k]==st[i]) return 0;
 return 1;}
int solutie() {return k==p;}
void tipar()
{int i,j; nr++;
 cout<<"Subgraful "<<nr<<endl<<"Nodurile:";
 for (i=1;i<=p;i++) cout<<st[i]<<" ";
 cout<<endl;
 cout<<"Muchiile: "; // cout<<"Arcele: "; pentru graful orientat
 for(i=1;i<=p;i++)
 for(j=i+1;j<=p;j++) // for(j=1;j<=p;j++) pentru graful orientat

```

```

    if (a[st[i]][st[j]]==1) cout<<st[i]<<"-"<<st[j]<<" ";
cout<<endl;
void bt() {//partea fixă a algoritmului backtracking}
void main()
{citere(); for(p=n;p>=1;p--) bt(); }

```

2. Verificarea dacă un graf G_s este subgraf al unui graf G .

Algoritmul. Se verifică dacă:

- numărul de noduri din graful G_s este mai mic sau cel mult egal cu numărul de noduri din graful G ;
- etichetele nodurilor din graful G_s există printre etichetele grafului G ;
- între nodurile din graful G_s există muchiile dintre nodurile din graful G și numai acelea.

Implementarea algoritmului. Se citesc, din două fișiere text *g1s.txt* și *g2s.txt*, informații despre două grafuri neorientate (orientate): de pe prima linie numărul de noduri și apoi, de pe următoarele rânduri, matricea de adiacență. În fișierul *g2s.txt* pe ultimul rând, după matricea de adiacență, este memorat un sir de numere care reprezintă etichetele nodurilor din acest graf. Matricele de adiacență ale celor două grafuri sunt $a1$ și $a2$, cu dimensiunea n , respectiv m . În vectorul v se memorează etichetele nodurilor celui de al doilea graf. Funcția *subgraf()* verifică dacă graful G_s este subgraf al grafului G .

```

#include<iostream.h>
int m,n,a1[10][10],a2[10][10],v[10];
fstream f1("g1s.txt",ios::in),f2("g2s.txt",ios::in);
int subgraf()
{
if (m>n) return 0;
else
    {for (int i=1;i<=m;i++)
     for(int j=1;j<=m;j++)
        if (a2[i][j]!=a1[v[i]][v[j]]) return 0;}
return 1;
}
void main()
{int i,j; f1>>n;
 for(i=1;i<=n;i++)
    for(j=1;j<=n;j++) f1>>a1[i][j]; f1.close();
f2>>m;
 for(i=1;i<=m;i++)
    for(j=1;j<=m;j++) f2>>a2[i][j];
 for(i=1;i<=m;i++) f2>>v[i]; f2.close();
if(subgraf()) cout<<"este subgraf ";
else cout<<"nu este subgraf";}

```

3. Verificarea dacă două matrice de adiacență pot reprezenta matricele de adiacență ale unui graf și ale unui subgraf al acestuia.

Algoritmul. Matricele de adiacență ale celor două grafuri au dimensiunea n , respectiv p . Se identifică graful care are ordinul mai mare (n) și gradul celuilalt graf – p ($n \geq p$). Pentru graful cu n noduri se generează toate matricele de adiacență ale subgrafurilor cu p noduri și se verifică dacă una dintre ele este identică cu matricea de adiacență a celuilalt graf. Generarea tuturor subgrafurilor cu p noduri se face cu metoda backtracking. În stivă se vor genera nodurile subgrafului.

Implementarea algoritmului. Se citesc din două fișiere text, *graf_1s.txt* și *graf_2s.txt*, informații despre matricele de adiacență a două grafuri neorientate: de pe prima linie,

numărul de noduri, apoi matricea de adiacență. Se verifică dacă unul dintre grafuri este subgraf al celuilalt, iar în caz afirmativ, se identifică care este graful și care este subgraful. Matricele de adiacență ale celor două grafuri sunt a_1 și a_2 . Variabila x se folosește pentru a identifica relația dintre cele două grafuri: dacă are valoarea 0, a două matrice de adiacență se asociază unui posibil subgraf, iar dacă are valoarea 1, prima matrice de adiacență se asociază unui posibil subgraf. Variabila logică $gasit$ se folosește pentru a determina dacă una dintre matricele de adiacență reprezintă matricea de adiacență a unui subgraf: are valoarea 0 – *False*, dacă nu este subgraf și valoarea 1 – *True* dacă este subgraf. În matricea b se va construi matricea de adiacență a unui subgraf format cu nodurile generate în stivă. Funcția $zero()$ se folosește pentru a inițializa cu 0 elementele matricei b – înainte de generarea fiecărui subgraf. Funcția $tipar()$ se folosește pentru a genera matricea b și pentru a o compara cu matricea asociată unui posibil subgraf. Funcția furnizează un rezultat logic: 1 – *true*, dacă matricea generată și matricea asociată unui posibil subgraf sunt egale; altfel, are valoarea 0 – *False*. Rezultatul acestei funcții este atribuit variabilei $gasit$. Subprogramul $bt()$ se execută atât timp cât nu s-a generat o matrice de adiacență egală cu cea asociată unui posibil subgraf ($gasit==0$) și se mai pot căuta soluții pentru matricea generată ($k>0$).

```
#include<iostream.h>
int n,p,a1[10][10],a2[10][10],b[10][10],v[10],nr,x,as,ev,k,gasit;
fstream f1("grafl_s.txt",ios::in),f2("graf2_s.txt",ios::in);
typedef int stiva[100];
stiva st;
void zero()
{for(int i=1;i<=p;i++)
    for(int j=1;j<=p;j++) b[i][j]=0;}
void init() {st[k]=0;}
int succesor()
{if (st[k]<n) {st[k]=st[k]+1; return 1;} else return 0;}
int valid()
{if (k>1 && st[k]<st[k-1]) return 0;
 for (int i=1;i<k;i++) if (st[k]==st[i]) return 0;
 return 1;}
int solutie() {return k==p;}
int tipar()
{int i,j; zero();
 if (x==0)
    {for(i=1;i<=p;i++)
        for(j=1;j<=p;j++) if (a1[st[i]][st[j]]==1) b[i][j]=1;
     for(i=1;i<=p;i++)
        for(j=1;j<=p;j++) if (a2[i][j]!=b[i][j]) return 0;}
    else
    {for(i=1;i<=p;i++)
        for(j=1;j<=p;j++) if (a2[st[i]][st[j]]==1) b[i][j]=1;
     for(i=1;i<=p;i++)
        for(j=1;j<=p;j++) if (a1[i][j]!=b[i][j]) return 0;}
 return 1;}
void bt(){//partea fixă a algoritmului backtracking }
void main()
{int i,j,m; f1>>n;
 for(i=1;i<=n;i++)
    for(j=1;j<=n;j++) f1>>a1[i][j]; f1.close();}
```

```

f2>>p;
for(i=1;i<=p;i++)
    for(j=1;j<=p;j++) f2>>a2[i][j]; f2.close();
if (p>n) {m=n; n=p; n=m; x=1;}
else x=0;
bt();
if (gasit)
    if (x) cout<<"prima matrice de adiacenta este a subgrafului";
    else cout<<"a doua matrice de adiacenta este a subgrafului";
else cout<<"nu este subgraf";}

```

Temă

1. Scrieți un program care citește din fișierul text *graf16.txt* matricea de adiacență a unui graf orientat – și care generează toate subgrafulurile care se pot obține. Informațiile despre subgrafuluri se vor salva în fișierul text *graf_s16.txt*. Pentru fiecare subgraf generat se va scrie, pe un rând, textul *Subgraful x are p noduri* (unde x este numărul de ordine al subgrafului generat, iar p numărul de noduri) și pe rândul următor textul *Arcele* – urmat de lista de arce a subgrafului generat.
2. Scrieți un program care citește din două fișiere text, *graf_1s.txt* și *graf_2s.txt* informații despre matricele de adiacență a două grafuri orientate (de pe prima linie numărul de noduri, apoi matricea de adiacență) și care verifică dacă unul dintre grafuri este subgraf al celuilalt. În caz afirmativ, se precizează care este graful și care este subgraful.
3. Scrieți un program care citește din fișierul text *graf16.txt* matricea de adiacență a unui graf orientat – și care generează subgraful care se obține prin eliminarea nodului care are cei mai mulți vecini.

2.7.6.3. Graful complementar

Fie graful $G = (X, U)$ și mulțimea de muchii (arce) V . Graful $G_c = (X, V)$ se numește **graf complementar** al grafului G dacă are proprietatea că două noduri sunt adiacente în graful G_c , numai dacă nu sunt adiacente în graful G ($U \cap V = \emptyset$).

Altfel spus, un **graf complementar** al grafului G conține aceleași noduri ca și graful G , dar nici o muchie din acest graf.

Exemple:

1. Pentru graful neorientat $G_1 = (X_1, U_1)$, definit anterior, graful $G_{1c} = (X_1, V_1)$, definit astfel:
 \rightarrow mulțimea nodurilor este $X_1 = \{1, 2, 3, 4, 5, 6, 7, 8\}$.
 \rightarrow mulțimea muchiilor este $V_1 = \{[1, 5], [1, 6], [1, 7], [2, 4], [2, 7], [5, 7], [5, 8]\}$.
Este graf complementar al grafului G_1 – figura 21.

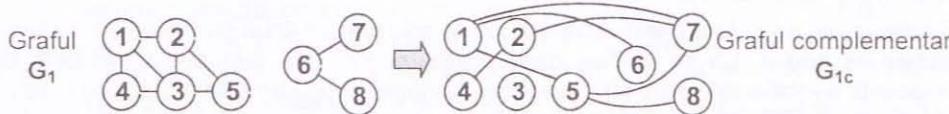


Fig. 21

2. Pentru graful orientat $G_{10} = (X_{10}, U_{10})$, definit anterior, graful $G_{10c} = (X_{10}, V_{10})$, definit astfel:
 \rightarrow mulțimea nodurilor este $X_{10} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.
 \rightarrow mulțimea arcelor este $V_{10} = \{[1, 3], [1, 5], [2, 4], [2, 8], [3, 2], [4, 2], [5, 2], [5, 6], [7, 1], [8, 2], [9, 3], [9, 10]\}$.

Este graf complementar al grafului G_{10} – figura 22.

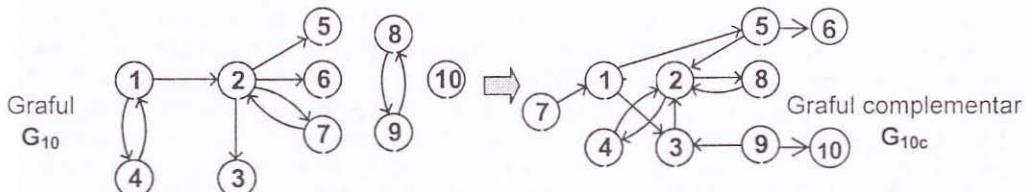


Fig. 22

Tema

Scripti un program care citește din două fișiere text *graf_1c.txt* și *graf_2c.txt* informații despre două grafuri neorientate – și varianta orientată – (de pe prima linie, numărul de noduri, apoi matricea de adiacență) și care verifică dacă unul dintre grafuri este graf complementar al celuilalt graf. (Indicație. Dacă cele două grafuri au același număr de noduri, se determină graful intersecție, care va trebui să fie graful vid).

2.7.6.4. Aplicații practice

1. Din grupul de n persoane între care s-au stabilit relații de prietenie, afișați, pentru fiecare persoană, persoanele cu care nu este în relație de prietenie. (Indicație. Se generează graful complementar.)
2. Din grupul de n persoane între care s-au stabilit relații de cunoștință, eliminați străinii de grup și singuraticii. (Indicație. Se generează un subgraf al grafului inițial, prin eliminarea tuturor nodurilor izolate sau terminale.)
3. La graful județelor, construiți un vector în care memorați, pentru fiecare județ, un indice pentru provinția istorică din care face parte (1 – Muntenia, 2 – Moldova etc.). Obțineți din graful județelor, subgrafurile provinciilor istorice. Precizați provinția istorică ce conține cele mai multe județe, și provinția istorică ce conține cele mai puține județe.
4. Din graful grotelor, obțineți subgraful grotelor care se găsesc la o înălțime h , cu $h_1 \leq h \leq h_2$. Verificați câte dintre aceste grote comunică direct cu exteriorul muntelui. Valorile pentru înălțimile h_1 și h_2 se citesc de la tastatură.
5. În rețeaua de străzi a orașului, două străzi se închid pentru a fi reparate. Etichetele intersecțiilor care sunt legate de aceste străzi se citesc de la tastatură. Să se verifice dacă, prin închiderea acestor străzi, traficul auto nu este perturbat, în sensul că vor exista intersecții la care nu se mai poate ajunge. (Indicație. Se generează graful parțial al traficului, prin eliminarea arcelor dintre nodurile precizate, și se verifică dacă există noduri care au gradul intern egal cu 0.)
6. Se citesc, din două fișiere text două matrice de adiacență a două grafuri care au același număr de noduri. Să se verifice dacă aceste matrice pot reprezenta matricele de adiacență a relației de vecinătate, respectiv a relației de prietenie pentru sătenii care au fânețe. (Indicație. Se verifică dacă graful cu mai puține muchii este graf parțial al celuilalt graf).

2.7.7. Conexitatea grafurilor

2.7.7.1. Lanțul

Într-un graf $G = (X, U)$ se definește **lanțul** ca fiind o succesiune de noduri care au proprietatea că, oricare ar fi două noduri succesive, ele sunt adiacente.

Graful neorientat

Dacă mulțimea nodurilor unui graf neorientat este $X = \{x_1, x_2, \dots, x_n\}$, un lanț de la nodul l_1 la nodul $l_k - L(l_1, l_k)$ – va fi definit prin mulțimea $L(l_1, l_k) = \{l_1, l_2, \dots, l_i, \dots, l_k\}$, unde $l_i \in X$ pentru orice i ($1 \leq i \leq k$), iar muchiile $[l_1, l_2], [l_2, l_3], [l_3, l_4], \dots, [l_{k-1}, l_k] \in U$. Lanțul poate fi interpretat ca un traseu prin care se parcurg anumite muchii ale grafului, traseul fiind ordinea în care se parcurg aceste muchii: $[l_1, l_2], [l_2, l_3], [l_3, l_4], \dots, [l_{k-1}, l_k]$. Fiecare pereche de noduri succesive din lanț reprezintă parcurgerea unei muchii. Dacă există $L(x_i, x_j)$, se spune că **nodul x_j este accesibil din nodul x_i** .

Graful orientat

Dacă mulțimea nodurilor unui graf orientat este $X = \{x_1, x_2, \dots, x_n\}$, un lanț de la nodul l_1 la nodul $l_k - L(l_1, l_k)$ – va fi definit prin mulțimea $L(l_1, l_k) = \{l_1, l_2, \dots, l_i, \dots, l_k\}$, unde $l_i \in X$, pentru orice i ($1 \leq i \leq k$). Arcele $[l_1, l_2], [l_2, l_3], [l_3, l_4], \dots, [l_{k-1}, l_k]$ au proprietatea că, oricare ar fi două arce succesive, ele au o extremitate comună. La definirea unui lanț, nu se ține cont de orientarea arcelor, ci numai de faptul că două noduri sunt legate de un arc.

Terminologie:

- **Lungimea unui lanț** reprezintă numărul de parcurgeri ale muchiilor, respectiv arcelor. De exemplu, lungimea lanțului $L(l_1, l_k)$ este **$k-1$** . Dacă o muchie (un arc) este parcursă de mai multe ori, se va număra fiecare dintre parcurgerile sale.
- **Lanțul de lungime minimă** dintre nodul x_i și nodul $x_j - L_{\min}(x_i, x_j)$ – este lanțul cu numărul minim de muchii (arce), din mulțimea nevidă de lanțuri $L(x_i, x_j)$.
- **Extremitățile unui lanț** sunt formate din nodul cu care începe și nodul cu care se termină lanțul (l_1 și l_k).
- **Sublanțul** este format dintr-un sir continuu de noduri din lanț. De exemplu, pentru lanțul $L(l_1, l_k) = \{l_1, l_2, \dots, l_i, \dots, l_j, \dots, l_k\}$, $L_s(l_i, l_j) = \{l_i, l_{i+1}, \dots, l_{j-1}, l_j\}$ este un sublanț al lanțului L .

Exemple:

- Pentru graful neorientat $G_{19} = (X_{19}, U_{19})$, definit astfel:

- **mulțimea nodurilor** este $X_{19} = \{1, 2, 3, 4, 5, 6, 7, 8\}$.
- **mulțimea muchiilor** este $U_{19} = \{[1, 2], [1, 5], [1, 6], [2, 3], [2, 5], [2, 6], [3, 4], [3, 6], [3, 7], [3, 8], [4, 8], [5, 6], [6, 7], [7, 8]\}$.

$L_1(1, 7) = \{1, 2, 3, 8, 4, 3, 6, 5, 2, 3, 7\}$ este un lanț între nodul cu eticheta 1 și nodul cu eticheta 7 (figura 23). Lungimea lanțului este 10.

- Pentru graful orientat $G_{20} = (X_{20}, U_{20})$, definit astfel:

- **mulțimea nodurilor** este $X_{20} = \{1, 2, 3, 4, 5, 6, 7\}$.
- **mulțimea arcelor** este $U_{20} = \{[1, 2], [1, 4], [2, 3], [2, 4], [3, 6], [3, 7], [4, 1], [4, 5], [5, 2], [5, 4], [6, 3], [6, 5], [7, 6]\}$.

$L_1(1, 5) = \{1, 2, 5, 6, 3, 6, 7, 6, 5\}$ este un lanț între nodul cu eticheta 1 și nodul cu eticheta 5 (figura 24). Lungimea lanțului este 8.

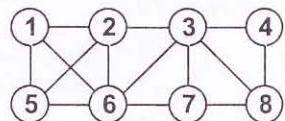


Fig. 23

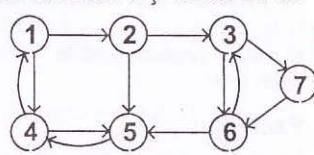
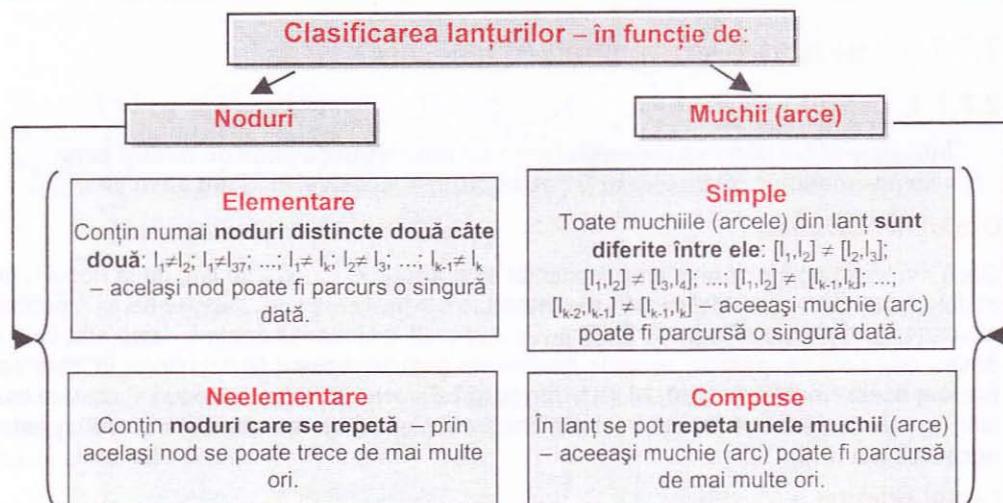


Fig. 24



Exemple:

1. Pentru graful neorientat G_{19} :

- Lanțul $L_1(1,7)$ definit anterior este un **lanț neelementar**, deoarece se repetă nodul cu eticheta 3. Este un **lanț compus**, deoarece în lanț se repetă muchia $[2,3]$.
- Lanțul $L_2(1,7) = \{1, 2, 3, 6, 7\}$ este un **lanț elementar** deoarece fiecare nod este parcurs o singură dată.
- Lanțul $L_3(1,7) = \{1, 6, 3, 2, 6, 7\}$ este un **lanț simplu**, deoarece nicio muchie nu se repetă, dar este un **lanț neelementar**, deoarece se repetă nodul cu eticheta 6.

2. Pentru graful orientat G_{20} :

- Lanțul $L_1(1,5)$ definit anterior este un **lanț neelementar**, deoarece se repetă nodul cu eticheta 6. Este un **lanț compus**, deoarece în lanț se repetă arcul $[6,7]$.
- Lanțul $L_2(1,5) = \{1, 2, 3, 7, 6, 5\}$ este un **lanț elementar**, deoarece fiecare nod este parcurs o singură dată.
- Lanțul $L_3(1,5) = \{1, 2, 4, 5, 2, 3, 6, 5\}$ este un **lanț simplu**, deoarece niciun arc nu se repetă, dar este un **lanț neelementar** deoarece se repetă nodul cu eticheta 2.

Teorema 10

Dacă un graf conține un lanț între două noduri, x și y , atunci conține un lanț elementar între nodurile x și y .

Demonstrație. Considerăm lanțul $L(x,y)=\{x, l_1, l_2, \dots, l_k, y\}$. Dacă lanțul nu este elementar, el conține cel puțin un nod z care se repetă – există i și j , astfel încât $l_i = l_j = z$: $L(x,y)=\{x, l_1, l_2, \dots, l_i, \dots, l_j, \dots, l_k, y\}$. Nodul z aparținând lanțului $L(x,y)$, înseamnă că el este accesibil din nodul x , iar nodul y este accesibil din nodul z . Înseamnă că din lanț se poate elimina sublanțul care leagă nodul z de el însuși: $L(x,y)=\{x, l_1, l_2, \dots, l_i, l_{i+1}, \dots, l_k, y\}$. În același mod, se poate elimina din lanț, toate sublanțurile care leagă un nod de el însuși, obținându-se în final un lanț în care fiecare nod nu apare decât o singură dată, adică un lanț elementar.

Exemplu:

În graful neorientat G_{19} , lanțul $L_1(1,7)$ este un lanț neelementar. Prin eliminarea din acest lanț a sublanțului $\{8, 4, 3\}$, se obține lanțul $\{1, 2, 3, 6, 5, 2, 3, 7\}$, care este un lanț neelementar. Prin eliminarea din acest lanț a sublanțului $\{6, 5, 2, 3\}$, se obține lanțul $\{1, 2, 3, 7\}$, care este un lanț elementar.

Teorema 11

Dacă un lanț este elementar, atunci este și lanț simplu.

Demonstrație – prin reducere la absurd. Presupunem că lanțul elementar este lanț compus. Dacă este lanț compus, el trebuie să parcurgă de două ori aceeași muchie (arc), ceea ce ar însemna să treacă de două ori prin nodurile adiacente muchiei (arcului). Lanțul fiind elementar, nu trece însă de două ori prin același nod.

Algoritmi pentru determinarea lanțurilor elementare într-un graf**1. Afisarea lanțurilor elementare dintre două noduri.**

Algoritm. Pentru generarea lanțurilor elementare între nodul x și nodul y , se folosește metoda **backtracking**. Nodurile lanțului elementar vor fi generate în stivă. Primul nod din stivă este nodul x . Condiția ca un nod adăugat în stivă să facă parte din soluție (să fie valid) este să formeze o muchie (un arc) cu nodul adăugat anterior în stivă și să nu mai existe în stivă (condiția ca lanțul să fie elementar). Soluția se obține atunci când nodul adăugat în stivă este nodul y .

Implementarea algoritmului. Se citește din fișierul text *graf1.txt* matricea de adiacență a unui graf neorientat, respectiv, din fișierul *graf2.txt* matricea de adiacență a unui graf orientat. Se mai citesc de la tastatură două valori numerice care reprezintă etichetele a două noduri din graf. Se caută toate lanțurile elementare care există între cele două noduri și se afișează. Dacă nu există nici un lanț elementar, se afișează un mesaj. Funcția *citeste()* se folosește pentru a citi matricea de adiacență din fișier. Variabila *este* se folosește pentru a verifica dacă s-a găsit un lanț elementar între cele două noduri (are valoarea 0 – *False*, dacă nu s-a găsit un lanț elementar). Pentru testarea programelor se folosesc graful neorientat G_1 și graful orientat G_9 .

```
#include<iostream.h>
typedef stiva[100];
int a[20][20],n,k,as,ev,x,y,este=0;
stiva st;
fstream f("graf1.txt",ios::in);
void citeste() {//se citește matricea de adiacență din fișier}
void init() {st[k]=0;}
int succesor()
{if (st[k]<n) {st[k]=st[k]+1; return 1;} else return 0;}
int valid()
{if(k>1) //condiția ca două noduri succesive să fie adiacente
 if (a[st[k-1]][st[k]]==0) return 0;
 //pentru graful orientat
 //if (a[st[k]][st[k-1]]==0 && a[st[k-1]][st[k]]==0) return 0;
 for (int i=1;i<k;i++)
 if (st[k]==st[i]) return 0;
return 1;}
int solutie() {return st[k]==y;}
//se obține soluția atunci când ultimul nod adăugat în stivă este y
void tipar ()
{for (int i=1,este=1;i<=k;i++) cout<<st[i]<<" "; cout<<endl;}
void bt() //partea fixă a algoritmului
{k=2; init();
 while (k>1)
 {as=1; ev=0;
 while(as && !ev)
```

```

    {as=succesor();
     if(as) ev=valid();}
    if(as)
        if (solutie()) tipar();
        else {k++; init();}
    else k--;}
void main()
{citere(); cout<<"x= "; cin>>x; cout<<"y= "; cin>>y; st[1]=x; bt();
 if (!este) cout<<"Nu exista lant";}

```

2. Afișarea tuturor lanțurilor elementare din graf.

Algoritmul. Pentru generarea tuturor lanțurilor elementare din graf, se vor genera lanțurile elementare dintre nodul **x** ($1 \leq x \leq n$) și nodul **y** ($1 \leq y \leq n$), folosind metoda **backtracking**, care se va apela pentru fiecare pereche de noduri **x** și **y** ($x \neq y$).

Implementarea algoritmului. Se citește din fișierul text *graf1.txt* matricea de adiacență a unui graf neorientat, respectiv, din fișierul *graf2.txt* matricea de adiacență a unui graf orientat. Se caută toate lanțurile elementare care există în graf și se afișează. Dacă nu există nici un lanț elementar, se afișează un mesaj. Pentru testarea programelor se folosesc graful neorientat G_1 și graful orientat G_9 .

```

#include<iostream.h>
typedef stiva[100];
int a[20][20],n,k,as,ev,x,y,este=0;
stiva st;
fstream f("graf1.txt",ios::in);
void citeste() {//la fel ca în exemplul precedent }
void init() //la fel ca în exemplul precedent }
int successor() //la fel ca în exemplul precedent }
int valid() //la fel ca în exemplul precedent }
int solutie() //la fel ca în exemplul precedent }
void tipar () //la fel ca în exemplul precedent }
void bt() //partea fixă a algoritmului - ca în exemplul precedent}
void main()
{citere();
 for (x=1;x<=n;x++)
    for (y=1;y<=n;y++) if (x!=y) {st[1]=x; bt();}
 if (!este) cout<<"Nu exista lanturi elementare";}

```

3. Verificarea unui sir de etichete de noduri dacă formează un lanț simplu.

Algoritmul. Se verifică dacă:

- sirul de etichete poate forma un lanț (dacă fiecare pereche de noduri consecutive din lanț este legată prin muchie, respectiv arc);
- lanțul este simplu (se verifică dacă muchiile formate cu nodurile din lanț nu se repetă).

Implementarea algoritmului. Se citesc din fișierul text *graf3.txt* lista muchiilor unui graf neorientat, respectiv, din fișierul *graf4.txt* lista arcelor unui graf orientat. Se citește apoi, de la tastatură, un sir de **k** numere, care reprezintă etichete ale nodurilor din graf. Se verifică dacă acest sir de noduri poate reprezenta un lanț simplu în graf. Sirul de numere citite de la tastatură se memorează în vectorul **v**. În vectorul **z** cu înregistrări de tip **muchie**, se memoră muchiile (arcele) pe care le formează două noduri succesive din sirul de numere. Pentru a identifica două muchii (arce) identice, perechile de etichete care formează o muchie sunt memorate ordonat. Funcția **citere()** se folosește pentru a citi informațiile din fișier și pentru a le memora în matricea de adiacență a grafului, funcția **lant()** se

folosește pentru a verifica dacă sirul de numere poate reprezenta un lanț din graf, funcția `simplu()` se folosește pentru a verifica dacă lanțul este un lanț simplu. Pentru testarea programelor se folosesc graful neorientat G_1 și graful orientat G_9 .

```
#include<iostream.h>
struct muchie {int x,y;};
muchie z[10];
int k,n,m,a[10][10],v[10];
fstream f("graf3.txt",ios::in);
void citeste()
{int i,x,y; f>>n>>m;
 for(i=1;i<=m;i++) {f>>x>>y; a[x][y]=1; a[y][x]=1;} f.close();}
int lant()
{for(int i=1;i<k;i++) if (a[v[i]][v[i+1]]==0) return 0;
 return 1;}
int simplu()
{for(int i=1;i<k;i++)
 for(int j=i+1;j<=k;j++) if(z[i].x==z[j].x && z[i].y==z[j].y) return 0;
return 1;}
void main()
{int i; citeste(); cout<<"k= ";cin>>k;
for(i=1;i<=k;i++) {cout<<"Eticheta "<<i<<" = "; cin>>v[i];}
for(i=1;i<k;i++) if (v[i]<v[i+1]) {z[i].x=v[i];z[i].y=v[i+1];}
else {z[i].x=v[i+1];z[i].y=v[i];}
if (lant())
 if (simplu()) cout<<"este lant simplu";
 else cout<<"nu este lant simplu";
else cout<<"nu este lant";}
```


Tema

1. Scrieți un program care citește dintr-un fișier text lista muchiilor unui graf neorientat (*graf3.txt*) sau variantă graf orientat (*graf4.txt*), și care:
 - a. Verifică dacă un sir de k numere citite de la tastatură reprezintă un lanț elementar pentru graf.
 - b. Caută toate lanțurile elementare cu lungimea d și le afișează. Valoarea numerică d se citește de la tastatură. Dacă nu există nici un lanț elementar cu lungimea d , se afișează un mesaj.
 - c. Caută toate lanțurile elementare, cu lungimea cea mai mică, ce există între două noduri x și y , și le afișează. Etichetele nodurilor se citesc de la tastatură. Dacă nu există nici un lanț elementar, se afișează un mesaj..
 - d. Caută toate lanțurile elementare între două noduri x și y , care trec printr-un nod z care are gradul minim în graf. Etichetele celor două noduri se citesc de la tastatură. Dacă nu există nici un lanț elementar, se afișează un mesaj.
 - e. Caută toate lanțurile elementare de lungime maximă și afișează câte lanțuri s-au găsit și care sunt ele. Dacă nu există nici un lanț elementar, se afișează un mesaj.
 - f. Caută și afișează cel mai lung lanț elementar care este format din noduri care au etichete cu numere consecutive, ordonate crescător.
 - g. Caută și afișează cel mai scurt lanț elementar care trece prin p noduri și prin q muchii (arce) date. Se citesc de la tastatură următoarele valori numerice: p – număr de noduri, un sir de p numere, care reprezintă etichete ale nodurilor din graf, q – număr de muchii și un sir de q perechi de numere, care reprezintă muchii (arce) din graf.
2. Într-un fișier text sunt memorate informații despre două grafuri neorientate G' și G'' : pe primul rând numărul de noduri ale celor două grafuri (n_1 și n_2), pe următoarele n_1

rânduri matricea de adiacență a grafului G' , iar pe următoarele n_2 rânduri matricea de adiacență a grafului G'' . Scrieți un program care citește din fișierul text informațiile despre cele două grafuri și care determină două noduri p și q (p din graful G' și q din graful G'') care – dacă se leagă printr-o muchie – asigură legătura, printr-un lanț de lungimea k , între două noduri precizate x și y (x din graful G' și y din graful G''). Valorile pentru x , y și k se citesc de la tastatură.

2.7.7.2. Ciclul

Un lanț care are toate muchiile distincte două câte două și extremități care coincid – se numește **ciclu**.

Ciclul este un **lanț simplu** ($[l_1, l_2] \neq [l_2, l_3]$; $[l_1, l_2] \neq [l_3, l_4]$; ...; $[l_1, l_2] \neq [l_{k-1}, l_k]$; ...; $[l_{k-2}, l_{k-1}] \neq [l_{k-1}, l_k]$), în care extremitățile coincid: $l_1 = l_k$.

Un graf fără cicluri se numește **graf aciclic**.

Dacă toate nodurile unui ciclu sunt distincte două câte două, cu excepția extremităților, ciclul se numește **ciclu elementar**.

Exemple:

1. Pentru graful neorientat G_{19} :

- Lanțul $L_4(1,1) = \{1, 2, 3, 6, 2, 1\}$ nu este un ciclu deoarece în lanț se repetă muchia $[1,2]$.
- Lanțul $L_5(1,1) = \{1, 2, 6, 3, 7, 6, 1\} = C_1$ este un ciclu neelementar deoarece se repetă nodul cu eticheta 6.
- Lanțul $L_6(1,1) = \{1, 2, 3, 7, 6, 1\} = C_2$ este un ciclu elementar deoarece nu se repetă niciun nod.

2. Pentru graful orientat G_{20} :

- Lanțul $L_4(1,1) = \{1, 2, 4, 5, 2, 1\}$ nu este un ciclu deoarece în lanț se repetă arcul $[1,2]$.
- Lanțul $L_5(1,1) = \{1, 2, 5, 6, 3, 2, 4, 1\} = C_1$ este un ciclu neelementar deoarece se repetă nodul cu eticheta 2.
- Lanțul $L_6(1,1) = \{1, 2, 3, 6, 5, 4, 1\}$ este un ciclu elementar deoarece nu se repetă niciun nod.

3. Un circuit electric poate fi reprezentat cu ajutorul unui graf orientat G_{21} , în care nodurile rețelei sunt nodurile grafului, iar sensul arcelor este dat de sensul ales pentru curentul electric – figura 25.

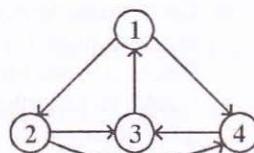
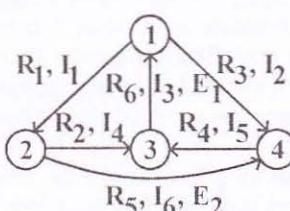
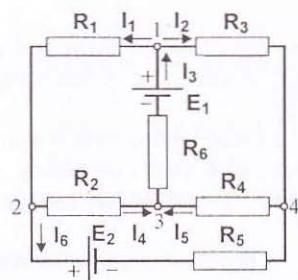


Fig. 25

Un ochi al rețelei electrice reprezintă un ciclu în graful orientat. Fiecare arc k are asociate trei mărimi: rezistența R_k , intensitatea curentului I_k și tensiunea electromotoare a sursei E_k . Semnul curentului electric este dat de sensul arcului: dacă arcul intră în nod, curentul electric are semnul plus, iar dacă arcul ieșe din nod, curentul electric are semnul minus.

Pentru fiecare nod din graf, se aplică teorema întâi a lui Kirchhoff:

$$\sum_{k=1}^p I_k = 0$$

unde p este suma dintre gradul intern și gradul extern ale nodului.

Pentru fiecare ciclu din graf, se aplică teorema a doua a lui Kirchhoff:

$$\sum_{k=1}^q E_k = \sum_{k=1}^q I_k \times R_k$$

unde q este numărul de arce ale ciclului.



Pentru circuitul electric din figura 25, se consideră cunoscute următoarele mărimi: $R_1, R_2, R_3, R_5, R_k, I_1, E_1$ și E_2 . Scrieți un program care să calculeze – în graful circuitului electric – mărimea rezistenței R_4 .

Teorema 12

Dacă un graf conține un ciclu, atunci conține și un ciclu elementar.

Demonstrație – la fel ca demonstrația de la Teorema 11, de la subcapitolul Lanțuri.

Două lanțuri – $L_1 = \{l_1, l_2, \dots, l_i, \dots, l_{k-1}, l_k, l_1\}$ și $L'_1 = \{l'_1, l'_2, \dots, l'_i, \dots, l'_{k-1}, l'_k, l'_1\}$ – care au aceeași lungime k , formează același ciclu dacă există un număr întreg j , astfel încât $l_i = l'_{(i+j) \bmod k+1}$, pentru orice $i=1, 2, 3, \dots, k$.

Exemplu:

În graful neorientat G_{19} lanțuri $L_1 = \{l_1, l_2, l_3, l_4, l_1\} = \{1, 2, 6, 5, 1\}$ și $L'_1 = \{l'_1, l'_2, l'_3, l'_4, l'_1\} = \{2, 6, 5, 1, 2\}$ care au lungimea 4, formează același ciclu. Se observă $l_1=1$ și $l_4=1$. Din $(1+j) \bmod 4 + 1 = 4$, rezultă că $j=2$. Verificăm dacă pentru acest j și pentru orice $i=2, 3, 4$, există două noduri cu aceeași etichetă în ambele lanțuri:

$$i=2 \Rightarrow (2+2) \bmod 4 + 1 = 1; \text{ dar: } l_2=2 \text{ și } l'_1=2 \Rightarrow l_2=l'_1.$$

$$i=3 \Rightarrow (3+2) \bmod 4 + 1 = 2; \text{ dar: } l_3=6 \text{ și } l'_2=6 \Rightarrow l_3=l'_2.$$

$$i=4 \Rightarrow (4+2) \bmod 4 + 1 = 3; \text{ dar: } l_4=5 \text{ și } l'_3=5 \Rightarrow l_4=l'_3.$$

În graful neorientat G_{19} lanțuri $L_1 = \{l_1, l_2, l_3, l_4, l_1\} = \{1, 2, 6, 5, 1\}$ și $L'_1 = \{l'_1, l'_2, l'_3, l'_4, l'_1\} = \{2, 5, 6, 1, 2\}$ care au lungimea 4, nu formează același ciclu. Se observă $l_1=1$ și $l_4=1$. Din $(1+j) \bmod 4 + 1 = 4$, rezultă că $j=2$. Verificăm dacă pentru acest j și pentru orice $i=2, 3, 4$, există două noduri cu aceeași etichetă în ambele lanțuri:

$$i=2 \Rightarrow (2+2) \bmod 4 + 1 = 1; \text{ dar: } l_2=2 \text{ și } l'_1=2 \Rightarrow l_2=l'_1.$$

$$i=3 \Rightarrow (3+2) \bmod 4 + 1 = 2; \text{ dar: } l_3=6 \text{ și } l'_2=5 \Rightarrow l_3 \neq l'_2.$$

Algoritm pentru determinarea ciclurilor elementare într-un graf

Algoritmul. Se generează toate lanțurile elementare care pornesc din nodul x ($1 \leq x \leq n$), trec prin cel puțin trei noduri și se pot închide cu nodul x . Pentru generarea acestor lanțuri elementare se folosește metoda backtracking, care se va apela pentru fiecare nod x . Soluția se obține atunci când nodul adăugat în stivă formează o muchie (un arc) cu nodu x – și stiva conține cel puțin 3 noduri.

Implementarea algoritmului. Se citește din fișierul text *graf1.txt* matricea de adiacență a unui graf neorientat, respectiv, din fișierul *graf2.txt* matricea de adiacență a unui graf orientat. Se caută toate ciclurile elementare care există în graf și se afișează. Dacă nu există nici un ciclu elementar, se afișează un mesaj. Pentru testarea programelor se folosesc graful neorientat G_1 și graful orientat G_9 .

```
#include <iostream.h>
typedef stiva[100];
int a[20][20],n,k,as,ev,x,este=0;
stiva st;
fstream f("grafl.txt",ios::in);
void citeste() { // se citește matricea de adiacență din fișier}
void init()
{ // este identică cu cea de la afișarea lanțurilor elementare}
int succesor()
{ // este identică cu cea de la afișarea lanțurilor elementare}
int valid()
{ // este identică cu cea de la afișarea lanțurilor elementare}
int solutie() {return a[st[k]][x]==1 && k>2;}
// se obține soluția atunci când ultimul nod adăugat este adiacent
// nodului x și în stivă există cel puțin trei noduri
void tipar ()
{for (int i=1,este=1;i<=k;i++) cout<<st[i]<<" ";
 cout<<x<<endl;}
void bt() { //partea fixă a algoritmului }
void main()
{citeste();
 for (x=1;x<=n;x++) {st[1]=x; bt();}
 if (!este) cout<<"Nu există nici un ciclu elementar";}

```

Temă

- În exemplul precedent, sunt afișate lanțuri care formează același ciclu (același ciclu este afișat de mai multe ori; deosebirea constă doar în nodul cu care se începe și se încheie ciclul). Modificați programul astfel încât să se eliminate afișarea informației redundante (afișarea aceluiași ciclu de mai multe ori).
- Scrieți un program care citește din fișierul text *graf3.txt* lista muchiilor unui graf neorientat – și afișează toate ciclurile elementare care trec printr-un nod cu gradul minim.
- Scrieți un program care citește din fișierul text *graf3.txt* lista muchiilor unui graf neorientat și care verifică dacă un sir de k numere citite de la tastatură reprezintă un **ciclu elementar** pentru graf.
- Scrieți un program care citește din fișierul text *graf4.txt* lista arcelor unui graf orientat și care:
 - Caută toate ciclurile elementare care există în graf și le afișează. Dacă nu există niciun ciclu elementar, se afișează un mesaj.
 - Caută toate ciclurile elementare care trec prin toate nodurile grafului și le afișează. Dacă nu există nici un ciclu elementar care trece prin toate nodurile grafului, se afișează un mesaj.
 - Caută toate ciclurile elementare de lungime maximă care trec prin nodul x – și le afișează. Eticheta nodului se citește de la tastatură. Dacă nu există nici un ciclu elementar care să treacă prin nodul x , se afișează un mesaj.

2.7.7.3. Drumul

Într-un **graf orientat** $G = (X, U)$ se definește un **drum** ca fiind o succesiune de noduri care au proprietatea că – oricare ar fi două noduri successive – ele sunt legate printr-un arc.

Dacă, într-un graf orientat, mulțimea nodurilor este $X = \{x_1, x_2, \dots, x_n\}$, iar mulțimea arcelor este $U = \{u_1, u_2, \dots, u_m\}$, un drum de la nodul d_1 la nodul d_k – $D(d_1, d_k)$ – va fi definit prin

mulțimea nodurilor $D(d_1, d_k) = \{d_1, d_2, \dots, d_k\}$, unde $d_i \in U$, pentru orice i ($1 \leq i \leq k$), iar arcele $[d_1, d_2], [d_2, d_3], [d_3, d_4], \dots, [d_{k-1}, d_k] \in U$. Drumul poate fi privit ca un lanț în care parcurgerea de la un nod la altul trebuie să se facă în sensul arcului care leagă nodurile. Dacă există $D(x_i, x_j)$, se spune că **nodul x_j este accesibil din nodul x_i** .

Terminologie:

- **Lungimea unui drum** este dată de numărul de arce care îl compun. În cazul în care arcele au asociate lungimi, lungimea unui drum este dată de suma lungimilor arcelor care îl compun.
- **Drumul de lungime minimă** dintre nodul d_i și nodul d_j – $D_{\min}(d_i, d_j)$ – este drumul cu numărul minim de arce din mulțimea nevidă de drumuri $D(d_i, d_j)$.
- **Subdrumul** este format dintr-un sir continuu de noduri din drum. De exemplu, pentru lanțul $D(d_1, d_k) = \{d_1, d_2, \dots, d_i, \dots, d_j, \dots, d_k\}$, $D_s(d_i, d_j)$ definit astfel: $D_s(d_i, d_j) = \{d_i, d_{i+1}, \dots, d_{j-1}, d_j\}$ – este un subdrum al drumului D .

Drumul elementar este drumul în care nodurile sunt distincte două câte două. **Drumul simplu** este drumul în care arcele sunt distincte două câte două.

Exemple – Pentru graful orientat G_{20} :

- Lanțul $L_7(1,6) = \{1, 2, 5, 6\}$ nu este un drum deoarece parcurgerea nu se face în sensul săgețiilor.
- Lanțul $L_8(1,6) = \{1, 2, 3, 6, 3, 6\} = D_1$ este un drum neelementar, deoarece se repetă eticheta nodurilor 3 și 6 – și compus, deoarece prin arcul $[3,6]$ s-a trecut de două ori.
- Lanțul $L_9(1,6) = \{1, 2, 3, 7, 6\} = D_2$ este un drum elementar, deoarece nu se repetă nici un nod.
- Lanțul $L_9(1,6) = \{1, 2, 4, 5, 2, 3, 6\} = D_3$ este un drum simplu, deoarece nici un arc nu a fost parcurs de două ori, dar este un drum neelementar, deoarece se repetă nodul cu eticheta 2.

Teorema 13

Dacă un graf neorientat conține un drum între două noduri, x și y , atunci, conține și un drum elementar, între nodurile x și y .

Demonstrație – la fel ca demonstrația de la Teorema 11 de la subcapitolul Lanțuri.

Algoritmi pentru determinarea drumurilor elementare într-un graf orientat

1. Afisarea drumurilor elementare din graf.

Algoritm. Se va folosi algoritmul pentru determinarea tuturor lanțurilor elementare din graf, la care elementul soluției generat în stivă, la nivelul k ($k > 1$), va fi considerat valid numai dacă trecerea de la nodul de pe nivelul $k-1$ la nodul de pe nivelul k se face, în graf, în sensul arcului.

Implementarea algoritmului. Se citește din fișierul *graf2.txt* matricea de adiacență a unui graf orientat. Se caută toate drumurile elementare care există în graf – și se afișează. Dacă nu există nici un drum elementar, se afișează un mesaj. Pentru testarea programului se folosește graful orientat G_9 .

```
#include <iostream.h>
typedef stiva[100];
int a[20][20], n, k, as, ev, x, este=0;
stiva st;
fstream f("graf2.txt",ios::in);
void citeste() { //se citeste matricea de adiacență din fisier}
```

```

void init() { //identică cu cea de la afişarea lanţurilor elementare}
int succesor(){//ca cea de la afişarea lanţurilor elementare}
int valid()
{if(k>1) if (a[st[k-1]][st[k]]==0) return 0;
 for (int i=1;i<k;i++) if (st[k]==st[i]) return 0;
return 1;}
int solutie() {return st[k]==y;}
void tipar ()
{for (int i=1,este=1;i<=k;i++) cout<<st[i]<<" ";
 cout<<x<<endl;}
void bt() { //partea fixa a algoritmului }
void main()
{citere();
 for (x=1;x<=n;x++)
 for (y=1;y<=n;y++)
 if (x!=y) {st[1]=x; bt();}
 if (!este) cout<<"Nu există";}

```

2. Algoritmul Roy-Warshall de determinare a matricei drumurilor.

Matricea drumurilor este o matrice pătrată binară de dimensiune n (n reprezentând ordinul grafului), definită astfel:

$$a[i][j] = \begin{cases} 1, & \text{dacă există drum de la nodul } i \text{ la nodul } j \\ 0, & \text{dacă nu există drum de la nodul } i \text{ la nodul } j \end{cases}$$

Informațiile din matricea drumurilor se pot folosi pentru a verifica dacă există drum între două noduri ale grafului

Algoritmul. Dacă nu există un arc de la nodul i la nodul j , considerăm că există drum de la nodul i la nodul j , dacă există un nod k ($k \neq i$ și $k \neq j$) care are proprietatea că există un drum de la nodul i la nodul k și un drum de la nodul k la nodul j . Matricea drumurilor se obține din matricea de adiacență prin transformări succesive, astfel: se generează toate nodurile k ($1 \leq k \leq n$), iar pentru fiecare nod k se generează toate perechile de noduri i ($i \neq k$) și j ($j \neq k$) și se verifică dacă $a[i][k]=1$ și $a[k][j]=1$. În caz afirmativ, în matricea de adiacență se atribuie valoarea 1 elementului $a[i][j]$.

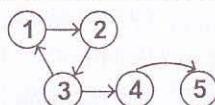


Fig. 26

G₂₂ Pentru graful G_{22} din figura 26 matricea de adiacență suferă următoarele cinci transformări. La fiecare transformare, dacă drumul de la nodul i la nodul j trece prin nodul intermediu k (drumul trece de la nodul i la nodul k și de la nodul k la nodul j), atunci elementului $a[i][j]$ îi se va atribui valoarea 1.

k=1					
1	2	3	4	5	
1	0	1	0	0	0
2	0	0	1	0	0
3	1	1	0	1	0
4	0	0	0	0	1
5	0	0	0	1	0

k=2					
1	2	3	4	5	
1	0	1	1	0	0
2	0	0	1	0	0
3	1	1	1	1	0
4	0	0	0	0	1
5	0	0	0	1	0

	1	2	3	4	5
1	0	1	0	0	0
2	0	0	1	0	0
3	1	1	0	1	0
4	0	0	0	0	1
5	0	0	0	1	0

k=4					
1	2	3	4	5	
1	1	1	1	1	0
2	1	1	1	1	0
3	1	1	1	1	0
4	0	0	0	0	1

	1	2	3	4	5
1	0	1	1	1	1
2	1	1	1	1	1
3	1	1	1	1	1
4	0	0	0	1	1

Interpretarea datelor din matricea obținută în urma transformărilor se face astfel: există drum de la nodul i la nodul j dacă $a[i][j]=1$. De exemplu, există drum de la nodul 2 la nodul 4, dar nu există drum de la nodul 4 la nodul 2.

Implementarea algoritmului. Se citește din fișierul *graf20.txt* matricea de adiacență a unui graf orientat. Se afișează toate perechile de noduri din graf între care există un drum. Funcția *citeste()* se folosește pentru a citi matricea de adiacență din fișier. Funcția *transforma()* se folosește pentru a transforma matricea de adiacență în matricea drumurilor. Matricea de adiacență va suferi n transformări succesive (pentru fiecare nouă valoare a lui k), astfel: fiecare element $a[i][j]$, cu $i \neq k$ și $j \neq k$, este înlocuit cu valoarea produsului $a[i][k] * a[k][j]$ – ceea ce este echivalent cu a atribui valoarea 1 acestui element dacă $a[i][k]=1$ și $a[k][j]=1$. Funcția *afiseaza()* se folosește pentru a afișa toate perechile de noduri i și j între care există drum. Aceste perechi sunt identificate în matricea drumurilor ca fiind indicele pentru linie și indicele pentru coloană ale elementelor care au valoarea 1. Pentru testarea programului se folosește graful orientat G_{22} .

```
#include <iostream.h>
int a[20][20],n;
fstream f("graf20.txt",ios::in);
void citeste() { //se citește matricea de adiacență din fișier}
void transforma()
{for(int k=1;k<=n;k++)
 for(int i=1;i<=n;i++)
   for(int j=1;j<=n;j++)
     if(a[i][j]==0 && i!=k && j!=k) a[i][j]=a[i][k]*a[k][j];}
void afiseaza()
{cout<<"Există drum între perechile de noduri"<<endl;
 for(int i=1;i<=n;i++)
   for(int j=1;j<=n;j++)
     if(a[i][j]==1 && i!=j) cout<< "("<<i<<","<<j<<")" << " ";
void main() {citeste(); transforma(); afiseaza();}
```

Complexitatea algoritmului Roy-Warshall

Algoritmul de transformare a matricei de adiacență în matricea drumurilor are ordinul de complexitate $O(n \times n \times n) = O(n^3)$, deoarece fiecare structură repetitivă *for* se execută de n ori, iar structurile *for* sunt imbricate.

Tema

Scriți un program care citește, din fișierul text *graf4.txt*, lista muchiilor unui graf orientat – și care:

- Verifică dacă un sir de k numere citite de la tastatură reprezintă un **drum elementar** pentru graf.
- Caută toate drumurile elementare cu lungimea cea mai mică dintre două noduri x și y și le afișează. Etichetele nodurilor se citesc de la tastatură. Dacă nu există niciun drum elementar, să se afișeze un mesaj.
- Caută toate drumurile elementare de lungime maximă și afișează câte drumuri s-au găsit și care sunt ele. Dacă nu există niciun drum elementar, să se afișeze un mesaj.

2.7.7.4. Circuitul

Într-un **graf orientat** un drum care are toate arcele distincte două câte două și extremități care coincid se numește **circuit**.

Circuitul este un **drum simplu** (arcele sunt distincte două câte două ($[d_1, d_2] \neq [d_2, d_3]$; $[d_1, d_2] \neq [d_3, d_4]$; ...; $[d_1, d_2] \neq [d_{k-1}, d_k]$; ...; $[d_{k-2}, d_{k-1}] \neq [d_{k-1}, d_k]$) în care extremitățile coincid ($d_1=d_k$).

Circuitul elementar este circuitul în care toate nodurile sunt distințe două câte două, cu excepția primului și a ultimului – care coincid.

Exemple – Pentru graful orientat G_{20} :

- Lanțul $L_{10}(1,1) = \{1, 2, 3, 6, 3, 6, 3, 7, 6, 5, 4, 1\}$ nu este un circuit, deoarece arcele $[3,6]$ și $[6,3]$ au fost parcurse de două ori.
- Lanțul $L_{11}(1,1) = \{1, 2, 3, 6, 3, 7, 6, 5, 2, 4, 1\} = C_1$ este un circuit neelementar, deoarece se repetă eticheta nodurilor 3 și 6.
- Lanțul $L_{12}(1,1) = \{1, 2, 3, 7, 6, 5, 4, 1\} = C_2$ este un circuit elementar, deoarece nu se repetă eticheta niciunui nod.

Teorema 14

Dacă un graf conține un circuit, atunci conține și un circuit elementar.

Demonstrație – la fel ca demonstrația de la Teorema 11 de la subcapitolul Lanțuri.

Algoritm pentru determinarea circuitelor elementare într-un graf orientat

Algoritmul. Se folosește același algoritm ca și în cazul determinării ciclurilor elementare într-un graf orientat, cu deosebirea că se ține cont de orientarea arcului care leagă două noduri.

```
#include <iostream.h>
typedef stiva[100];
int a[20][20],n,k,as,ev,x,este=0;
stiva st;
fstream f("graf2.txt",ios::in);
void citeste() { //se citește matricea de adiacență din fișier}
void init() { //identică cu cea de la afișarea lanțurilor elementare}
int succesor() { //identică cu cea de la afișarea lanțurilor elementare}
int valid()
{if(k>1) if (a[st[k-1]][st[k]]==0) return 0;
 for (int i=1;i<k;i++) if (st[k]==st[i]) return 0;
return 1;}
int solutie() {return a[st[k]][x]==1 && k>1;}
//se obține soluția atunci când ultimul nod adăugat este adiacent
//nodului x și în stivă există cel puțin două noduri
void tipar ()
{for (int i=1,este=1;i<=k;i++) cout<<st[i]<<" ";
 cout<<x<<endl;}
void bt() { //partea fixă a algoritmului }
void main() {citeste();
    for (x=1;x<=n;x++) {st[1]=x; bt();}
    if (!este) cout<<"Nu există circuite"; }
```

Tema



Scrieți un program care citește, din fișierul text *graf4.txt*, lista arcelor unui graf orientat – și care:

- Caută toate circuitele elementare care există în graf și le afișează.
Dacă nu există niciun circuit elementar, se afișează un mesaj.
- Caută toate circuitele elementare de lungime maximă care trec prin un nod **x** și le afișează. Eticheta nodului se citește de la tastatură. Dacă nu există niciun circuit elementar care să treacă prin nodul **x**, se afișează un mesaj.
- Caută toate circuitele elementare care trec prin toate nodurile grafului și le afișează.
Dacă nu există niciun circuit elementar care trece prin toate nodurile grafului, se afișează un mesaj.

- d. Caută toate circuitele elementare care trec numai prin noduri care au gradul intern egal cu gradul extern – și le afișează. Dacă nu există niciun circuit elementar, se afișează un mesaj.

2.7.7.5. Graful conex

Un graf G se numește **graf conex** dacă are proprietatea că, pentru orice pereche de noduri diferite între ele, există un lanț care să le lege.

Altfel spus, un graf este conex dacă, pentru orice pereche de noduri $\{x_i, x_j\}$, care au proprietatea $x_i \neq x_j$ există un lanț de la x_i la x_j .

Exemple:

1. Graful neorientat $G_{19} = (X_{19}, U_{19})$ definit anterior este un graf conex, deoarece oricare ar fi două noduri din mulțimea X_x , ele pot fi legate printr-un lanț. De exemplu, $(1,2) \Rightarrow \{1,2\}$ sau $\{1,5,2\}$; $(1,3) \Rightarrow \{1,2,3\}$ sau $\{1,6,3\}$; $(1,3) \Rightarrow \{1,2,3\}$ sau $\{1,6,3\}$; $(1,4) \Rightarrow \{1,2,3,4\}$ sau $\{1,6,7,4\}$; $(1,5) \Rightarrow \{1,5\}$ sau $\{1,2,6,5\}$ etc.
2. Graful orientat $G_{20} = (X_{20}, U_{20})$ definit anterior este un graf conex, deoarece oricare ar fi două noduri din mulțimea X_x , ele pot fi legate printr-un lanț. De exemplu, $(1,2) \Rightarrow \{1,2\}$ sau $\{1,4,2\}$; $(1,3) \Rightarrow \{1,2,3\}$ sau $\{1,2,5,6,3\}$; $(1,3) \Rightarrow \{1,2,3\}$ sau $\{1,2,5,6,3\}$; $(1,4) \Rightarrow \{1,4\}$ sau $\{1,2,5,4\}$; $(1,5) \Rightarrow \{1,2,5\}$ sau $\{1,4,5\}$ etc.
3. Graful neorientat $G_{23} = (X_{23}, U_{23})$ – figura 27 – definit astfel:
 → mulțimea nodurilor este $X_{23} = \{1, 2, 3, 4, 5, 6, 7, 8\}$.
 → mulțimea muchiilor este $U_{23} = \{[1,2], [1,4], [2,3], [2,4], [3,4], [5,6], [5,7]\}$.

nu este conex, deoarece nu există nici un lanț între un nod din mulțimea $C_1 = \{1, 2, 3, 4\}$ și un nod din mulțimea $C_2 = \{5, 6, 7\}$, sau din mulțimea $C_3 = \{8\}$.

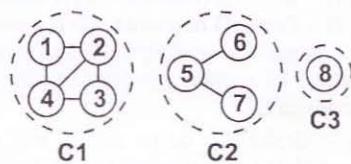


Fig. 27

Dacă un graf $G = (X, U)$ nu este conex, se poate defini un **subgraf conex** al grafului G , adică se poate defini o mulțime $X' \subset X$ care să inducă subgraful $G' = (X', U')$, ce are proprietatea că este conex.

Dacă un graf $G = (X, U)$ nu este conex, se numește **componentă conexă a grafului** un subgraf conex al său $C = (X', U')$, maximal în raport cu această proprietate (conține numărul maxim de noduri din G care au proprietatea că sunt legate cu un lanț).

Altfel spus, subgraful conex C este o componentă conexă a grafului dacă are proprietatea că nu există nici un lanț, al grafului G , care să unească un nod x_i al subgrafului C ($x_i \in X'$) cu un nod x_j care nu aparține subgrafului C ($x_j \in X - X'$).

Observație: Un graf conex are o singură componentă conexă (graful însuși).

Exemplu – În graful neorientat G_{23} definit anterior, fiecare dintre cele trei mulțimi de noduri, C_1 , C_2 și C_3 , induce câte o componentă conexă.

Dacă un graf este definit prin mulțimea nodurilor și mulțimea muchiilor,

- mulțimea nodurilor este $X_{23} = \{1, 2, 3, 4, 5, 6, 7, 8\}$.
- mulțimea muchiilor este $U_{23} = \{[1,2], [1,4], [2,3], [2,4], [3,4], [5,6], [5,7]\}$.

pentru identificarea componentelor conexe, procedați astfel:

- PAS1.** Se formează prima componentă conexă, pornind de la prima muchie. Se scriu nodurile incidente cu muchia în prima componentă conexă – $C_1 = \{1, 2\}$ – și se marchează muchia: [1,2]. Componenta conexă C_1 este componenta curentă.

- PAS2.** Se parcurg celelalte muchii. Dacă se găsește o muchie nemarcată, care conține noduri din componenta conexă curentă, se adaugă nodurile la componentă: $C_1=\{1,2,4,3\}$, și se marchează muchia: [1,4], [2,3], [2,4], [3,4].
- PAS3.** Dacă mai există muchii nemarcate, se identifică prima muchie nemarcată și se formează următoarea componentă conexă, scriind nodurile incidente cu muchia – $C_2=\{5,6\}$ – și se marchează muchia: [5,6]. Componenta conexă C_2 este componenta curentă.
- PAS4.** Se parcurg celelalte muchii. Dacă se găsește o muchie nemarcată care conține noduri din componenta conexă curentă, se adaugă nodurile la componentă: $C_2=\{5,6,7\}$ și se marchează muchia: [5,7].
- PAS5.** Se execută Pasul 3 și Pasul 4 până se marchează toate muchiile, identificând următoarele componente conexe.
- PAS6.** Se verifică dacă toate nodurile din mulțimea nodurilor se regăsesc într-o componentă conexă. Dacă există noduri care nu aparțin unei componente conexe, acestea sunt noduri izolate, și vor forma, fiecare dintre ele, o componentă conexă. $C_3=\{8\}$.

Teorema 15

Numărul minim de muchii m_{\min} necesare pentru ca un graf neorientat, cu n noduri, să fie conex este $n-1$.

Altfel spus, într-un graf neorientat conex cu m muchii și n noduri: $m \geq n-1$.

Demonstrație. Notăm cele două propozitii, astfel:

(1) – Graful G neorientat, cu n noduri, are $n-1$ muchii și este conex.

(2) – Graful G neorientat, cu n noduri, este conex și minimal cu această proprietate (dacă se înlătură orice muchie din graf, el devine neconex).

Trebuie să demonstrăm că (1) \Rightarrow (2) – Se folosește principiul inducției matematice (se notează cu P_i propoziția i):

P_1 – Graful G_1 , cu un nod și nici o muchie ($m=1-1=0$), este conex și este minimal cu această proprietate (nu mai există muchii care să fie eliminate).

P_2 – Graful G_2 , cu două noduri și o muchie ($m=2-1=1$), este conex (muchia nu poate lega decât cele două noduri) și este minimal cu această proprietate (prin eliminarea muchiei, nodurile sunt izolate obținându-se două componente conexe).

P_3 – Graful G_3 , cu trei noduri și două muchii ($m=3-1=2$), este conex (cele două muchii sunt incidente; unul dintre noduri este adjacent cu cele două muchii – și prin el va trece lanțul care leagă celelalte două noduri) și este minimal cu această proprietate (prin eliminarea unei muchii, unul dintre noduri este izolat, obținându-se două componente conexe).

P_n – Graful G_n , cu n noduri și $n-1$ muchii, este conex și minimal cu această proprietate (se presupune adevărată).

P_{n+1} – Considerând propoziția P_n adevărată, trebuie să demonstrăm că graful G_{n+1} , cu $n+1$ noduri și $n+2$ muchii, este conex și minimal cu această proprietate. Prin adăugarea unui nod și a unei muchii la graful G_n , se poate obține un graf conex. Nodul adăugat se leagă cu muchia nouă de graful G_n (și, implicit, la lanțurile din acest graf), obținându-se un nou graf conex. Presupunem că graful obținut nu este minimal cu această proprietate. Înseamnă că, prin eliminarea unei muchii, se obține un graf conex. Dacă eliminăm muchia adăugată, se izolează nodul $n+1$, obținându-se două componente conexe. Dacă se elimină o muchie din graful G_n , se obțin două componente conexe, deoarece graful G_n este minimal cu această proprietate. Rezultă că graful G_n este minimal cu această proprietate.

Exemple de grafuri conexe cu n noduri și număr minim de muchii:

→ Fiecare nod este legat de alte două, cu excepția a două noduri terminale. Există $n-2$ noduri cu gradul 2 și două noduri cu gradul 1:

$$m_{\min} = \frac{2 \times (n-2) + 2 \times 1}{2} = n-1$$

→ Un nod este legat de celelalte $n-1$ noduri. Există un nod cu gradul $n-1$ și $n-1$ noduri cu gradul 1:

$$m_{\min} = \frac{1 \times (n-1) + (n-1) \times 1}{2} = n-1$$

Propoziția 4. Dacă un graf cu n noduri are p componente conexe, atunci numărul minim de muchii care trebuie adăugate, ca să devină conex, este $p-1$.

Demonstrație. Putem considera fiecare componentă conexă ca un nod al unui graf cu p noduri. Numărul minim de muchii necesare pentru ca acest graf să fie conex este $p-1$.

Propoziția 5. Dacă un graf conex cu n noduri are $n-1$ muchii, atunci orice pereche de noduri este legată printr-un lanț, și numai unul.

Demonstrație – prin reducere la absurd. Graful G fiind conex, înseamnă că există cel puțin un lanț care leagă oricare două noduri, x și y . Presupunem că există cel puțin două lanțuri între nodurile x și y : L_1 și L_2 . Înseamnă că, suprimând o muchie din lanțul al doilea, graful rămâne conex, deoarece nodurile x și y vor fi legate prin lanțul L_1 . Înseamnă că un graf cu n noduri și $n-2$ muchii este conex, ceea ce contrazice Teorema 15. Rezultă că cele două noduri, x și y , nu sunt legate decât printr-un singur lanț.

Propoziția 6. Dacă un graf neorientat cu n noduri și m muchii este conex, numărul maxim de muchii care se pot elimina pentru a obține un **graf parțial conex** este: $m-n+1$.

Demonstrație. Deoarece numărul minim de muchii necesare pentru ca un graf să fie conex este $n-1$, atunci din graf se pot elimina restul muchiilor: $m-n+1$.

Teorema 16

Un graf neorientat **conex** cu n noduri și $n-1$ muchii este **aciclic** și maximal cu această proprietate.

Demonstrație: Notăm cele două propoziții, astfel:

- (1) – Graful G neorientat, cu n noduri, este conex și are $n-1$ muchii.
- (2) – Graful G neorientat, cu n noduri, este aciclic și maximal cu această proprietate.

Trebuie să demonstrăm că (1)⇒(2).

- 1) **Este aciclic** – prin reducere la absurd. Presupunem că acest graf conține cel puțin un ciclu, $C=\{x_1, x_2, x_3, \dots, x_i, x_1\}$. Înseamnă că, dacă vom înlătura din graf muchia $[x_1, x_2]$, se obține un graf conex – nodul va fi legat de toate celelalte noduri din ciclu prin lanțul $L=\{x_2, x_3, \dots, x_i, x_1\}$ – ceea ce contrazice teorema anterioară, că un graf cu n noduri și $n-1$ muchii este conex și minimal cu această proprietate.
- 2) **Este maximal cu această proprietate** (dacă se adaugă o nouă muchie, graful nu mai este aciclic) – prin reducere la absurd. Presupunem că, prin adăugarea unei muchii oarecare $[x,y]$, se obține un graf aciclic. Graful fiind conex, înseamnă că între nodurile x și y există un lanț $L(x,y)=\{x, \dots, z, \dots, y\}$, iar prin adăugarea muchiei $[x,y]$ lanțul se închide, formând un ciclu $C=\{x, \dots, z, \dots, y, x\}$ – ceea ce contrazice presupunerea că graful este aciclic.

Propoziția 7. Dacă un graf neorientat conex are n noduri și m muchii, numărul de muchii care trebuie eliminate, pentru a obține un **graf parțial conex aciclic**, este egal cu $m-n+1$.

Demonstrație. Din Propoziția 6, rezultă că – înlăturând din graf $m-n+1$ muchii – se obține un graf parțial conex, iar din Teorema 16 rezultă că acest graf este aciclic. Graful fiind conex, înseamnă că între oricare două noduri, x_i și x_j , există un lanț elementar și numai unul (Propoziția 5). Orice nouă muchie $[x_i, x_j]$ adăugată va forma cu acest lanț un ciclu elementar.

Propoziția 8. Dacă un graf are n noduri, m muchii și p componente conexe, numărul de muchii care trebuie eliminate, pentru a obține un **graf parțial aciclic**, este egal cu $m-n+p$.

Demonstrație. Fiecare componentă conexă i ($1 \leq i \leq p$) va avea n_i noduri și m_i muchii, iar:

$$\sum_{i=1}^p n_i = n \text{ și } \sum_{i=1}^p m_i = m.$$

Numărul de muchii care trebuie eliminate din componenta conexă i , pentru a obține un graf parțial conex aciclic, este egal cu $m_i - n_i + 1$ (Propoziția 7). Rezultă că numărul total de muchii care trebuie eliminate din graf, pentru a obține graf parțial conex aciclic, este egal cu:

$$\sum_{i=1}^p (m_i - n_i + 1) = \sum_{i=1}^p m_i - \sum_{i=1}^p n_i + p = m - n + p$$

Propoziția 9. Pentru a obține, dintr-un graf neorientat conex, două componente conexe, numărul minim de muchii care trebuie înălțurate m_{\min} este egal cu gradul minim din graf: $m_{\min} = \text{grad}_{\min}$.

Demonstrație. Cele două componente conexe ale unui graf neorientat conex $G=(X,U)$ se obțin astfel:

- Componenta conexă $C_1=(X_1,U_1)$ se formează dintr-un nod izolat x_i – pentru a elimina un număr minim de muchii, nodul se alege dintre nodurile care au gradul minim: $X_1=\{x_i\}$ și $U_1=\emptyset$.
- Componenta conexă $C_2=(X_2,U_2)$ se formează din restul nodurilor din graf: $X_2=X-\{x_i\}$ și $\text{card}(U_2)=m-\text{grad}_{\min}$.

Teorema 17

Numărul maxim de muchii m_{\max} dintr-un graf neorientat, cu **n noduri și p componente conexe**, este:

$$m_{\max} = \frac{(n-p) \times (n-p+1)}{2}$$

Demonstrație – prin reducere la absurd. Numărul maxim de muchii corespunde unor $p-1$ componente conexe formate din noduri izolate și o componentă conexă formată cu $n-p+1$ noduri. Pentru ca să se obțină numărul maxim de muchii, ultima componentă trebuie să fie un graf complet K_{n-p+1} care are numărul de muchii m :

$$m = \frac{(n-p) \times (n-p+1)}{2}$$

Să presupunem că există însă un graf format din $p-2$ componente conexe formate din noduri izolate și cu două componente conexe C_1 cu n_1 noduri și m_1 muchii și C_2 cu n_2 noduri și m_2 muchii, cu $n_1 \geq n_2 \geq 2$, $n_1 + n_2 = n-p+2$ care are numărul maxim de muchii $m_1 + m_2 = m_{\max}$. Cele două componente conexe pentru a avea un număr maxim de muchii trebuie să fie grafuri complete: $C_1=K_{n_1}$ și $C_2=K_{n_2}$. Modificăm acest graf, mutând un nod din componenta C_2 în componenta C_1 . În componenta C_1 acest nod trebuie unit prin muchii cu cele n_1 noduri ale componentei. Numărul de muchii ale componentei C_1 va fi $m_1 + n_1$. În componenta C_2 prin eliminarea nodului, se înălțură și cele $n_2 - 1$ muchii care îl uneau cu celelalte noduri din graf. Numărul de muchii ale componentei C_2 va fi $m_2 - n_2 + 1$. Numărul total de muchii al noului graf este $m_1 + n_1 + m_2 - n_2 + 1 \geq m_1 + m_2 + 1 = m_{\max} + 1$. Concluzia contrazice ipoteza că graful inițial avea numărul maxim de muchii. Înseamnă că graful care are numărul maxim de muchii este cel care conține $p-1$ componente conexe formate din noduri izolate și o componentă conexă formată cu $n-p+1$ noduri care este graful complet K_{n-p+1} .

2.7.7.6. Graful tare conex

Un **graf orientat** G se numește **graf tare conex** dacă are proprietatea că, pentru orice pereche de noduri diferite între ele, există un drum care să le lege.

Altfel spus, un graf orientat este tare conex dacă – pentru orice pereche de noduri $\{x_i, x_j\}$, care au proprietatea $x_i \neq x_j$ – există un drum de la x_i la x_j .

Exemple:

1. Graful orientat $G_{20} = (X_{20}, U_{20})$, definit anterior, este un graf tare conex, deoarece există un circuit elementar care trece prin toate nodurile grafului: {1,2,3,7,6,5,4,1}; altfel spus, oricare ar fi două noduri din mulțimea X_{20} , ele pot fi legate printr-un drum.

2. Graful orientat $G_{24} = (X_{24}, U_{24})$ – figura 28 – definit astfel:

→ mulțimea nodurilor este $X_{24} = \{1, 2, 3, 4, 5, 6\}$.

→ mulțimea arcelor este $U_{24} = \{[1,2], [1,4], [2,3], [3,1], [4,5], [5,4]\}$.

nu este conex, deoarece nu există nici un drum între un nod din mulțimea $C_2 = \{4, 5\}$ și un nod din mulțimea $C_1 = \{1, 2, 3\}$, sau din mulțimea $C_3 = \{6\}$.

Dacă un graf orientat $G = (X, U)$ nu este tare conex, se poate defini un **subgraf tare conex** al grafului G , adică se poate defini o mulțime $X' \subseteq X$ care să inducă subgraful $G' = (X', U')$, ce are proprietatea că este tare conex.

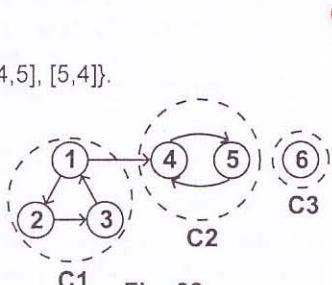


Fig. 28

Dacă un graf $G = (X, U)$ nu este conex, se numește **componentă tare conexă a grafului** un subgraf conex $C = (X', U')$ al său, maximal în raport cu această proprietate (conține numărul maxim de noduri din G care au proprietatea că sunt legate printr-un drum).

Altfel spus, subgraful tare conex C este o componentă tare conexă a grafului dacă are proprietatea că nu există nici un drum al grafului G care să unească un nod x_i al subgrafului C ($x_i \in X'$) cu un nod x_j care nu aparține subgrafului C ($x_j \in X - X'$).

Exemplu – În graful orientat G_{24} definit anterior, fiecare dintre cele trei mulțimi de noduri – C_1 , C_2 și C_3 – induce câte o componentă tare conexă.

Observație: **Un graf tare conex are o singură componentă tare conexă** (graful însuși).

Terminologie:

→ **Subgraful predecesorilor unui nod** este format din acel nod și din mulțimea nodurilor din care este accesibil nodul.

→ **Subgraful succesorilor unui nod** este format din acel nod și din mulțimea nodurilor care sunt accesibile din el.

Observație: **Componenta tare conexă** din care face parte un nod este dată de **intersecția dintre subgraful predecesorilor și subgraful succesorilor** aceluia nod.

Dacă un graf este definit prin mulțimea nodurilor și mulțimea muchiilor,

→ mulțimea nodurilor este $X_{24} = \{1, 2, 3, 4, 5, 6\}$.

→ mulțimea arcelor este $U_{24} = \{[1,2], [1,4], [2,3], [3,1], [4,5], [5,4]\}$.

pentru identificarea componentelor tare conexe procedați astfel:

PAS1. Se identifică subgraful succesorilor primului nod din primul arc (în exemplu, nodul 1), folosind algoritmul de determinare a unei componente conexe: $S_1 = \{1, 2, 3, 4, 5\}$.

PAS2. Se identifică subgraful predecesorilor primului nod din primul arc (în exemplu, nodul 1), folosind algoritmul de determinare a unei componente conexe, în care se iau în calcul numai nodurile care apar în arc în a doua poziție: $P_1 = \{1, 2, 3\}$.

PAS3. Se determină componenta tare conexă – prin intersecția celor două mulțimi de noduri: $C_1 = S_1 \cap P_1 = \{1, 2, 3\}$.

PAS4. Se identifică, în mulțimea arcelor, primul nod care nu face parte din componenta tare conexă evidențiată anterior (în exemplu, nodul 4) și se reiau **Pasul 1**, **Pasul 2** și **Pasul 3**, pentru a determina subgraful succesorilor, respectiv subgraful predece-

sorilor nodului și apoi următoarea componentă conexă, prin intersecția celor două multimi (în exemplu, nodul 4 și $S_2 = \{4, 5\}$, $P_2 = \{1, 2, 3, 4, 5\}$ și $C_2 = S_2 \cap P_2 = \{4, 5\}$). Se repetă acest pas până când nu se mai identifică în mulțimea arcelor niciun nod care să nu aparțină unei componente tare conexe.

- PAS5.** Se verifică dacă toate nodurile din mulțimea nodurilor se regăsesc într-o componentă tare conexă. Dacă există noduri care nu aparțin unei componente tare conexe, acestea sunt noduri izolate – și vor forma, fiecare dintre ele o componentă conexă. $C_3 = \{6\}$.

Graful componentelor tare conexe ale unui graf care nu este tare conex $G = (X, U)$ se obține prin reducerea fiecărei componente conexe la un nod.

Identificarea grafului componentelor tare conexe este utilă în rezolvarea unor probleme, prin descompunerea problemei în subprobleme (rezolvarea problemei pentru fiecare componentă tare conexă) și combinarea soluțiilor – strategia **divide et impera**.

Exemplu:

G₂₅ În graful orientat $G_{25} = (X_{25}, U_{25})$ – figura 29 – definit astfel:

- mulțimea nodurilor este $X_{25} = \{1, 2, 3, 4, 5, 6, 7\}$.
- mulțimea arcelor este $U_{25} = \{[1, 2], [1, 4], [2, 3], [3, 1], [4, 5], [5, 6], [6, 4], [7, 5], [7, 3]\}$.

G_{c25} componentele tare conexe sunt $C_1 = \{1, 2, 3\}$, $C_2 = \{4, 5, 6\}$ și $C_3 = \{7\}$, iar graful componentelor tare conexe $G_{c25} = (X_{c25}, U_{c25})$ – figura 30 – este definit astfel.

- mulțimea nodurilor este $X_{c25} = \{C_1, C_2, C_3\}$.
- mulțimea arcelor este $U_{c25} = \{[C_1, C_2], [C_3, C_1], [C_3, C_2]\}$.

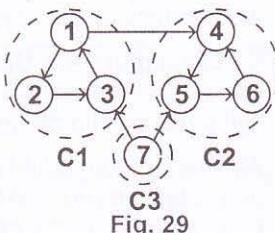


Fig. 29

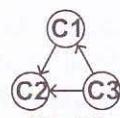


Fig. 30

Algoritmi pentru determinarea conexității grafurilor

1. Determinarea componentelor conexe într-un graf.

Algoritmul. O componentă conexă este formată din toate nodurile între care există un lanț elementar. Deoarece un nod nu poate să aparțină decât unei componente conexe, se va folosi un vector cu n elemente – pentru a memora nodurile care au fost deja înregistrate într-o componentă conexă. Inițial, elementele vectorului au valoarea 0 (nici un nod nu a fost înregistrat într-o componentă conexă), iar dacă un nod x va fi adăugat la o componentă conexă, valoarea elementului x din vector va fi 1. Pentru a **verifica dacă există un lanț elementar între două noduri x și y** se folosește **metoda backtracking**.

Observație. Acest algoritm poate fi folosit atât în grafurile neorientate, cât și în grafurile orientate. Mai poate fi folosit și pentru a verifica dacă un graf este conex: fie se verifică dacă între nodul 1 și fiecare dintre celelalte noduri ale grafului există un lanț elementar, fie se verifică dacă prima componentă conexă obținută conține toate nodurile grafului.

Implementarea algoritmului. Se citește din fișierul *graf19.txt* matricea de adiacență a unui graf neorientat (varianta din fișierul *graf20.txt* – matricea de adiacență a unui graf orientat) și se afișează componentele conexe. În vectorul v se memorează nodurile care au fost deja înregistrate într-o componentă conexă. Variabila **este** se folosește pentru a verifica dacă s-a găsit un lanț elementar între două noduri (are valoarea 0 – *False*, dacă nu s-a găsit un lanț elementar). Variabila **m** se folosește pentru a număra componentele conexe identificate. Funcția **citeste()** se folosește pentru a citi matricea de adiacență din fișier. Funcția **componente()** se folosește pentru a afișa componentele conexe ale grafului: pentru fiecare

nod x ($1 \leq x \leq n$) care nu a fost afişat într-o componentă conexă ($v[x] = 0$), se caută toate nodurile y ($1 \leq y \leq n$ și $y \neq x$) care sunt legate cu un lanț elementar de nodul x . Pentru un nod y găsit, se marchează în vectorul v faptul că a fost adăugat la o componentă conexă ($v[y] = 1$). Pentru testarea programului se folosesc graful neorientat G_4 și graful neorientat G_{22} .

```
#include <iostream.h>
typedef stiva[100];
int a[20][20], n, k, as, ev, x, y, v[20], este=0;
stiva st;
fstream f("gr8af19.txt", ios::in);
void citeste() { //se citește matricea de adiacență din fișier}
void init() { //identică cu cea de la afișarea lanțurilor elementare}
int succesor() { //identică cu cea de la afișarea lanțurilor elementare}
int valid() { //identică cu cea de la afișarea lanțurilor elementare}
int solutie() { return st[k]==y; }
//se obține soluția atunci când ultimul nod adăugat în stivă este y
void tipar() { este=1; }
//dacă există soluție (un lanț între nodurile x și y),
//variabila este va avea valoarea 1
void bt() { //identică cu cea de la afișarea lanțurilor elementare }
void componente()
{int m=0;
for(x=1;x<=n;x++)
    if (v[x]==0)
        {m++; st[m]=x; v[x]=1;
        cout<<"Componenta conexă "<<m<<": "<<x<<" ";
        for(y=1;y<=n;y++)
            if (x!=y) {este=0; bt();
                        if (este) {v[y]=1; cout<<y<<" ;}}
        cout<<endl;}}
void main() {citeste(); componente();}
```

2. Determinarea componentelor conexe într-un graf orientat.

Algoritmul. O componentă conexă este formată din toate nodurile între care există un drum, cel puțin de la un nod la altul (drumul nu trebuie să asigure legătura în ambele sensuri). Altfel spus, dacă există un drum de la un nod x la un nod y , cele două noduri x și y aparțin aceleiași componente conexe – chiar dacă nu există și un drum de la nodul y la nodul x . Și în acest algoritm se folosește un vector cu n elemente pentru a memora nodurile care au fost deja înregistrate într-o componentă conexă. Pentru a **verifica dacă există un drum de la nodul x la nodul y** se folosește matricea **drumurilor**.

Pentru graful G_{22} din figura 26, în matricea drumurilor se observă că există drum de la nodul 1 la oricare celelalte patru noduri din graf. Graful are o singură componentă conexă.

Implementarea algoritmului. Se citește din fișierul *graf20.txt* matricea de adiacență a unui graf orientat și se determină dacă graful este conex. Dacă nu este, se afișează componentele conexe. În vectorul v se memorează nodurile care au fost deja înregistrate într-o componentă conexă. Funcția *citeste()* se folosește pentru a citi matricea de adiacență în fișier. Funcția *transformă()* se folosește pentru a transforma matricea de adiacență în matricea drumurilor. Funcția *componente()* se folosește pentru a afișa componentele conexe ale grafului: pentru fiecare nod i ($1 \leq i \leq n$) care nu a fost afișat într-o componentă conexă

	1	2	3	4	5
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	1	1	1
4	0	0	0	1	1
5	0	0	0	1	1

$(v[i]=0)$, se caută toate nodurile j ($1 \leq j \leq n$ și $j \neq i$) la care ajunge un drum ce pornește din nodul i . Pentru un nod j găsit, se marchează în vectorul v faptul că a fost adăugat la o componentă conexă ($v[j]=1$). Pentru testarea programului se folosește graful orientat G_{22} .

```
#include <iostream.h>
typedef stiva[100];
int a[20][20],n,v[20];
fstream f("graf20.txt",ios::in);
void citeste() { //se citește matricea de adiacență din fișier}
void transforma()
{for(int k=1;k<=n;k++)
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
            if(a[i][j]==0 && i!=k && j!=k) a[i][j]=a[i][k]*a[k][j];}
void componente()
{int i,j,m=0;
 for(int i=1;i<=n;i++)
    if (v[i]==0)
        {m++; v[i]=1; cout<<"Componenta conexă "<<m<<": "<<i<<" ";
         for(int j=1;j<=n;j++)
             if(a[i][j]==1 && i!=j) {v[j]=1; cout<<j<<" ";}
         cout<<endl;}}
void main() {citeste(); transforma(); componente();}
```

3. Determinarea componentelor tare conexe într-un graf orientat.

Algoritmul. O componentă tare conexă este formată din toate nodurile i și j între care există un drum elementar de la nodul i la nodul j , dar și de la nodul j la nodul i . Se folosește un vector cu n elemente, pentru a memora nodurile care au fost deja înregistrate într-o componentă tare conexă. Inițial, elementele vectorului au valoarea 0 (nici un nod nu a fost înregistrat într-o componentă conexă), iar dacă un nod i va fi adăugat la o componentă conexă, valoarea elementului i din vector va fi 1. Pentru a **verifica dacă există un drum elementar de la nodul i la nodul j** , se pot folosi două variante ale algoritmului:

Varianta 1. Se folosește metoda **backtracking**.

Implementarea algoritmului. Se citește din fișierul *graf20.txt* matricea de adiacență a unui graf orientat – și se determină dacă graful este tare conex. Dacă nu este, se afișează componentele tare conexe. Variabila *este1* se folosește pentru a verifica dacă s-a găsit un drum elementar de la nodul i la nodul j (are valoarea 0 – *False*, dacă nu s-a găsit un drum elementar), variabila *este2* se folosește pentru a verifica dacă s-a găsit un drum elementar de la nodul j la nodul i (are valoarea 0 – *False*, dacă nu s-a găsit un drum elementar), iar variabila *este* se folosește pentru a verifica dacă s-a găsit un drum elementar de la nodul x la nodul y (are valoarea 0 – *False*, dacă nu s-a găsit un drum elementar). În vectorul v se memorează nodurile care au fost deja înregistrate într-o componentă tare conexă. Variabila *m* se folosește pentru a număra componentele conexe identificate. Funcția *citeste()* se folosește pentru a citi matricea de adiacență din fișier. Funcția *componente()* se folosește pentru a afișa componentele tare conexe ale grafului: pentru fiecare nod i ($1 \leq i \leq n$) care nu a fost afișat într-o componentă conexă ($v[i]=0$), se caută toate nodurile j ($1 \leq j \leq n$ și $j \neq i$) care sunt legate cu un drum elementar care pornește din nodul i , dar și de la care pornește un drum care ajunge în nodul i . Pentru un nod j găsit, se marchează în vectorul v faptul că a fost adăugat la o componentă tare conexă ($v[j]=1$). Pentru testarea programului se folosește graful orientat G_{22} .

```

#include <iostream.h>
typedef stiva[100];
int a[20][20],n,k,as,ev,x,y,v[20],estel,este2,este=0;
stiva st;
fstream f("graf19.txt",ios::in);
void citeste()//se citește matricea de adiacență din fișier}
void init() //identică cu cea de la afișarea drumurilor elementare}
int succesor() //identică cu cea de la afișarea drumurilor elementare}
int valid() //identică cu cea de la afișarea drumurilor elementare}
int solutie() {return st[k]==y;}
//se obține soluția atunci când ultimul nod adăugat în stivă este y
void tipar() {este=1;}
//dacă există soluție (un drum între nodurile x și y),
//variabila este va avea valoarea 1
void bt() //identică cu cea de la afișarea drumurilor elementare }
void componente()
{int i,j,m=0;
for(i=1;i<=n;i++)
  if (v[i]==0)
    {m++; v[i]=1; cout<<"Componenta conexă "<<m<<": "<<i<<" ";
     for(j=1;j<=n;j++)
       if (j!=i)
         {x=i; y=j; st[l]=x; este=0; bt(); estel=este;
          x=j; y=i; st[l]=x; este=0; bt(); este2=este;
           if (estel && este2) (v[j]=1; cout<<j<<" ");
         cout<<endl;})
void main() {citeste(); componente();}

```

Varianta 2. Se folosește matricea drumurilor. Se determină două matrice ale drumurilor: una corespunzătoare grafului G și alta corespunzătoare unui graf în care toate arcele au sensul invers decât în graful G (G_t – **graful transpus** al grafului G). Din intersecția celor două matrice, se va obține o matrice în care vor fi evidențiate numai drumurile care există în ambele sensuri, între oricare două noduri i și j .

Pentru graful G_{22} din figura 26, prin inversarea arcelor se obține graful transpus G_{22t} din figura 31. Matricele de adiacență și matricele drumurilor pentru cele două grafuri și matricea intersecție sunt prezentate mai jos.

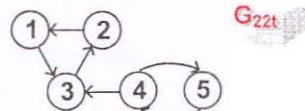


Fig. 31

Matricea de adiacență G_{22}					
	1	2	3	4	5
1	0	1	0	0	0
2	0	0	1	0	0
3	1	0	0	1	0
4	0	0	0	0	1
5	0	0	0	1	0

Matricea drumurilor G_{22}					
	1	2	3	4	5
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	1	1	1
4	0	0	0	1	1
5	0	0	0	1	1

Matricea de adiacență G_{22t}					
	1	2	3	4	5
1	0	0	1	0	0
2	1	0	0	0	0
3	0	1	0	0	0
4	0	0	1	0	1
5	0	0	0	1	0

Matricea drumurilor G_{22t}					
	1	2	3	4	5
1	1	1	1	0	0
2	1	1	1	0	0
3	1	1	1	0	0
4	1	1	1	1	1
5	1	1	1	1	1

Matricea intersecție

	1	2	3	4	5
1	1	1	1	0	0
2	1	1	1	0	0
3	1	1	1	0	0
4	0	0	0	1	1
5	0	0	0	1	1

Interpretarea datelor din matricea intersecție se face astfel: două noduri i și j fac parte din aceeași componentă tare conexă – dacă există drum de la nodul i la nodul j ($a[i][j]=1$). Pe matricea intersecție se pot identifica cele două componente conexe: $C_1=\{1,2,3\}$ și $C_2=\{4,5\}$.

Implementarea algoritmului. Se citește din fișierul *graf20.txt* matricea de adiacență a a unui graf orientat. Se folosește matricea ap în care se memorează arcele în sens invers și matricea drumurilor obținută din aceasta. Pentru a identifica nodurile i și j între care există un drum atât de la nodul i la nodul j , cât și de la nodul j la nodul i , se obține matricea b care este intersecția celor două matrice ale drumurilor: a și ap . Funcția *citeste()* se folosește pentru a citi matricea de adiacență din fișier. Funcția *transforma_s()* se folosește pentru a transforma matricea de adiacență a în matricea drumurilor, iar funcția *transforma_p()* pentru a transforma matricea de adiacență ap în matricea drumurilor. Funcția *intersectie()* se folosește pentru a obține intersecția celor două matrice ale drumurilor. Funcția *componente()* se folosește pentru a afișa componentele conexe ale grafului: pentru fiecare nod i ($1 \leq i \leq n$) care nu a fost afișat într-o componentă conexă ($v[i]=0$), se caută în matricea intersecție a drumurilor b toate nodurile j ($1 \leq j \leq n$ și $j \neq i$) la care ajunge un drum ce pornește din nodul i . Pentru un nod j găsit, se marchează în vectorul v faptul că a fost adăugat la o componentă tare conexă ($v[j]=1$). Pentru testarea programului se folosește graful orientat G_{22} .

```
#include <iostream.h>
typedef stiva[100];
int a[20][20],n,v[20],ap[10][10],b[10][10],nr;
fstream f("graf19.txt",ios::in);
void citeste() { //se citește matricea de adiacență din fișier}
void predecesor()
{for(int i=1;i<=n;i++)
    for (int j=1;j<=n;j++) ap[i][j]=a[j][i];}
void transforma_s()
{for(int k=1;k<=n;k++)
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
            if(a[i][j]==0 && i!=k && j!=k) a[i][j]=a[i][k]*a[k][j];}
void transforma_p()
{for(int k=1;k<=n;k++)
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
            if(ap[i][j]==0 && i!=k && j!=k) ap[i][j]=ap[i][k]*ap[k][j];}
void intersectie()
{for(int i=1;i<=n;i++)
    for (int j=1;j<=n;j++) b[i][j]= ap[i][j]*ap[i][j];}
void componente()
{for(int i=1;i<=n;i++)
    if (v[i]==0)
        {nr++; v[i]=1; cout<<endl<<"Componenta "<<nr<<": "<<i<<" ";
        for (int j=1;j<=n;j++)
            if (b[i][j]==1 && i!=j) {cout<<j<<" "; v[j]=1;}}}
void main()
{citeste(); predecesor(); transforma_s(); transforma_p();
intersectie(); componente();}
```

Temă

1. Scrieți un program care citește din fișierul text *graf3.txt* (*graf4.txt*) lista muchiilor (arcelor) unui graf neorientat (orientat) – și care verifică dacă graful este conex și afișează un mesaj de informare.
2. Scrieți un program care citește din fișierul text *graf14.txt* lista de adiacență a unui graf orientat – și care verifică dacă graful este tare conex și afișează un mesaj de informare.
3. Comparați, din punct de vedere al eficienței, cei doi algoritmi de determinare a componentelor tare conexe dintr-un graf orientat.
4. Scrieți un program care citește din fișierul text *graf2.txt* matricea de adiacență a unui graf orientat – și care determină graful componentelor tare conexe și numărul minim de arce care trebuie adăugate pentru ca graful să devină tare conex. Soluțiile pentru arcele care trebuie adăugate se vor salva în fișierul text *graf_comp.txt*. **Indicație:** Cele p componente tare conexe vor fi noduri în noul graf – și vor fi reprezentate printr-un vector cu p elemente de tip înregistrare:

```
struct comp {int n;
             int v[10];};
comp c[10];
```

iar arcele vor fi reprezentate prin matrice de adiacență. O componentă conexă, *c[i]* are *c[i].n* noduri; un nod *j* al componentei *i* fiind *c[i].v[j]*.

2.7.7.7. Aplicații practice

1. Din grupul de *n* persoane între care s-au stabilit relații de prietenie și de vecinătate, determinați grupurile de prieteni, grupurile de vecini și grupurile de vecini prieteni. (**Indicație.** Se determină componentele conexe pentru fiecare graf.)
2. Folosind informațiile din graful județelor determinați numărul minim, respectiv numărul maxim, de județ prin care trebuie să treacă un turist care tranzitează prin România, știind că vine din Ungaria și trebuie să ajungă în Bulgaria. (**Indicație.** Se determină un lanț de lungime minimă, respectiv maximă, între două județe limitrofe, care sunt la granița cu cele două țări.)
3. Din graful grotelor stabiliți grotile care vor fi inundate de un pârâu subteran care izvorăște dintr-o groă a cărei etichetă se citește de la tastatură. Precizați tunelul (sau tunelurile) prin care pârâul va ajunge în exteriorul muntelui. (**Indicație.** O groă este inundată dacă se găsește la același nivel sau la un nivel mai jos decât o groă în care există apă. Se transformă graful neorientat într-un graf orientat, în care arcele vor pune în evidență relația: grota *x* este în relație cu grota *y* dacă apa curge de la grota *x* la grota *y*. În acest graf determinați toate drumurile care pornesc din grota din care izvorăște pârâul).
4. În rețeaua de străzi a orașului, patru străzi se închid pentru a fi reparate. Etichetele intersecțiilor care sunt legate de aceste străzi se citesc de la tastatură. Să se verifice dacă prin închiderea acestor străzi nu se izolează unele zone ale orașului de alte zone. (**Indicație.** Se generează graful parțial al traficului, prin eliminarea arcelor dintre nodurile precizate și se verifică dacă graful obținut este tare conex.)

2.7.8. Parcurgerea grafului

Parcurgerea grafului reprezintă operația prin care sunt examineate în mod sistematic nodurile sale, pornind de la un nod dat *i*, astfel încât fiecare nod accesibil din nodul *i* pe muchiile (arcele) adiacente să fie atins o singură dată. **Vecinii unui nod** sunt reprezentați de toate **nodurile adiacente lui și accesibile din el**.

Vizitarea sau traversarea unui graf este operația prin care se parcurge graful trecându-se de la un nod i (nodul curent) la nodurile vecine lui, într-o anumită ordine, în vederea prelucrării informațiilor asociate nodurilor. Pentru parcurgerea grafurilor, există următoarele două metode:

1. **Metoda de parcurgere „în lățime” – Breadth First (BF).** Se vizitează mai întâi un nod inițial i , apoi vecinii acestuia, apoi vecinii nevizitați ai acestora, și așa mai departe – până când se parcurg toate nodurile grafului.
2. **Metoda de parcurgere „în adâncime” – Depth First (DF).** Se vizitează mai întâi un nod inițial i , după care se parcurge primul dintre vecinii săi nevizitați, de exemplu j , după care se trece la primul vecin nevizitat al lui j , și așa mai departe – până când se parcurge în adâncime ramura respectivă. Când s-a ajuns la capătul ei, se revine la nodul din care s-a plecat ultima dată, și se parcurge următorul său vecin nevizitat.

G₂₆. Exemplu – pentru graful neorientat G₂₆ = (X₂₆, U₂₆) – figura 32 – definit astfel:

→ multimea nodurilor este X₂₆ = {1, 2, 3, 4, 5, 6, 7, 8, 9}.

→ multimea muchiilor este U₂₆ = {[1, 2], [1, 3], [1, 4], [2, 5], [3, 5], [3, 6], [3, 7], [4, 7], [5, 8], [6, 8], [6, 9]}.

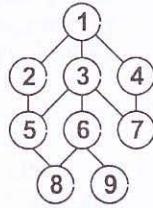


Fig. 32

Graful G₂₆ poate fi parcurs astfel, pornind din nodul 1:

1. În cazul metodei de parcurgere BF – „în lățime”, vor fi parcurse pe rând nodurile:

1, 2, 3, 4, 5, 6, 7, 8, 9

2. În cazul metodei de parcurgere DF – „în adâncime”, vor fi parcurse pe rând nodurile:

1, 2, 5, 3, 6, 8, 9, 7, 4

2.7.8.1. Parcurgerea în lățime – Breadth First

Pentru această metodă de parcurgere a unui graf cu n noduri, se folosesc următoarele structuri de date și date elementare:

1. **Variabilele de memorie:** n pentru numărul de noduri ale grafului, k pentru nodul care se prelucreză, i și j pentru parcurgerea matricei de adiacență și a vectorilor.
2. **Matricea de adiacență a grafului** a .
3. **Vectorul de vizitare vizitat.** El conține n elemente în care se înregistrează nodurile vizitate. Elementele vectorului vizitat[i] sunt definite astfel:

$$\text{vizitat}[i] = \begin{cases} 1, & \text{dacă nodul } i \text{ a fost vizitat} \\ 0, & \text{dacă nodul } i \text{ nu a fost vizitat încă} \end{cases}$$

4. **Coadă de aşteptare c a nodurilor parcurse.** Coda de aşteptare c gestionează nodurile prelucrate. Pentru coada de aşteptare se folosesc implementarea statică cu un vector c cu n elemente. Pentru gestionarea cozii de aşteptare se folosesc două variabile de memorie **prim** și **ultim** care joacă rolul de indicatori pentru a memora adresa primului element din listă (capul cozii) și, respectiv, adresa ultimului element din listă (coada cozii). În această coadă de aşteptare vor fi înregistrate nodurile vizitate, în ordinea în care au fost vizitate. Inițial, coada de aşteptare conține un singur element care corespunde nodului cu care începe parcurgerea grafului, iar valoarea indicatorilor **prim** și **ultim** este 1. Pe măsură ce se parcurge graful, se completează următoarele elemente ale vectorului c . Atunci când se prelucreză un nod k de la capul cozii de aşteptare, prin coada cozii de aşteptare se introduc toate nodurile i vecine cu nodul k care nu au fost vizitate încă.

Algoritmul pentru parcurgerea grafului este următorul:

PAS1. Se citesc din fișier, valoarea pentru variabila n și matricea de adiacență a grafului.

PAS2. Se initializează cu 0 elementele vectorului de vizitare **vizitat**, deoarece inițial nici unul dintre noduri nu a fost vizitat.

PAS3. Se introduce de la tastatură valoarea variabilei de memorie **k**, corespunzătoare primului nod cu care începe parcurgerea grafului.

PAS4. Se initializează coada de așteptare **c** (indicatorii și conținutul primului element introdus în coada de așteptare), astfel: $\text{prim} \leftarrow 1$, $\text{ultim} \leftarrow 1$ și $c[\text{prim}] \leftarrow k$, deoarece, în coada de așteptare **c** este înregistrat doar nodul **k** cu care începe parcurgerea.

PAS5. Se actualizează vectorul **vizitat** – atribuind elementului **k** al vectorului valoarea 1, deoarece a fost vizitat (**vizitat[k]←1**).

PAS6. Cât timp coada nu este vidă ($\text{prim} <= \text{ultim}$) execută

PAS7. Se extrage următorul element din coada de așteptare, corespunzător nodului căruia i se vor vizita vecinii ($k \leftarrow c[\text{prim}]$;).

PAS8. Pentru toate nodurile i vecine ale vârfului **k**, nevizitate încă, execută

PAS9. Se incrementează valoarea indicatorului **ultim**, pentru a înregistra următorul nod care va trebui vizitat prin adăugare la coada cozii de așteptare ($\text{ultim} \leftarrow \text{ultim} + 1$;).

PAS10 Se adaugă la sfârșitul cozii de așteptare **c**, nodul i vecin nodului **k** care nu a fost încă vizitat ($c[\text{ultim}] \leftarrow i$;).

PAS11 Prin adăugarea nodului **j** la coada de așteptare **c** se consideră că acest nod a fost vizitat – și se actualizează elementul din vectorul de vizitare care corespunde acestui nod (**vizitat[i]←1** ;). Se revine la Pasul 6.

PAS12 Se pregătește indicatorul **prim** pentru a se extrage următorul nod din coada de așteptare, ai cărui vecini vor fi vizitați ($\text{prim} \leftarrow \text{prim} + 1$;). Se revine la Pasul 7.

PAS13 Se afișează lista nodurilor vizitate, în ordinea în care au fost vizitate, prin extragerea lor din vectorul **c**, pornind de la elementul 1 și până la elementul **n**.

Pentru graful G_{26} , pornind din nodul 1, nodurile în coada de așteptare sunt adăugate astfel:

vizitat	1	0	0	0	0	0	0	0	0	0
	1	2	3	4	5	6	7	8	9	10
c	1									

Initializarea cozii de așteptare cu primul nod: $\text{prim}=1$; $\text{ultim}=1$.

vizitat	1	1	1	1	0	0	0	0	0	0
	1	2	3	4	5	6	7	8	9	10
c	1	2	3	4						

Prelucrarea nodului: $\text{prim}=1 \Rightarrow \text{ultim}=4$.

vizitat	1	1	1	1	1	0	0	0	0	0
	1	2	3	4	5	6	7	8	9	10
c	1	2	3	4	5					

Prelucrarea nodului: $\text{prim}=2 \Rightarrow \text{ultim}=5$.

vizitat	1	1	1	1	1	1	1	0	0	0
	1	2	3	4	5	6	7	8	9	10
c	1	2	3	4	5	6	7			

Prelucrarea nodului: $\text{prim}=3 \Rightarrow \text{ultim}=7$.

vizitat	1	1	1	1	1	1	1	0	0
	1	2	3	4	5	6	7	8	9
c	1	2	3	4	5	6	7		10
Prelucrarea nodului: prim=4 \Rightarrow ultim=7.									
vizitat	1	1	1	1	1	1	1	1	0
	1	2	3	4	5	6	7	8	9
c	1	2	3	4	5	6	7	8	10
Prelucrarea nodului: prim=5 \Rightarrow ultim=8.									
vizitat	1	1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9
c	1	2	3	4	5	6	7	8	9
Prelucrarea nodului: prim=6 \Rightarrow ultim=9.									
vizitat	1	1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9
c	1	2	3	4	5	6	7	8	9
Prelucrarea nodului: prim=7 \Rightarrow ultim=9.									
vizitat	1	1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9
c	1	2	3	4	5	6	7	8	9
Prelucrarea nodului: prim=8 \Rightarrow ultim=9.									
vizitat	1	1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9
c	1	2	3	4	5	6	7	8	9
Prelucrarea nodului: prim=9 \Rightarrow ultim=9.									
vizitat	1	1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9
c	1	2	3	4	5	6	7	8	9
Prelucrarea nodului: prim=10 \Rightarrow ultim=9. Terminare: prim>ultim									

Pentru implementarea algoritmului se folosesc subprogramele:

- funcția procedurală **citește** creează matricea de adiacență, prin preluarea informațiilor din fișierul **f** (dimensiunea matricei de adiacență și matricea);
- funcția procedurală **init** initializează coada de așteptare cu primul nod vizitat;
- funcția operand **este_vida** testează coada de așteptare dacă este vidă;
- funcția procedurală **adauga** adaugă un nod la coada de așteptare;
- funcția procedurală **elimina** elimină nodul prelucrat din coada de așteptare;
- funcția procedurală **prelucrare** prelucrează primul nod din coadă: adăugă la coada de așteptare toți vecinii nevizitați ai acestui nod și apoi îl elimină din coada de așteptare;
- funcția procedurală **afisare** afișează nodurile grafului în ordinea prelucrării.

Pentru testarea programului, se folosește graful neorientat G_{26} , a cărui matrice de adiacență se citește din fișierul text **graf21.txt**.

```
int n,a[10][10],vizitat[20],c[20],prim,ultim,k;
fstream f("graf21.txt",ios::in);
void citește() { //se citește matricea de adiacență din fișier}
void init(int k) {prim=ultim=1; c[ultim]=k; vizitat[k]=1;}
```

```

int este_vida() {return ultim<prim;}
void adauga(int i) {ultim++; c[ultim]=i; vizitat[i]=1;}
void elibera() {prim++;}
void prelucrare()
{int i; k=c[prim];
 for (i=1;i<=n;i++) if (a[k][i]==1 && vizitat[i]==0) adauga(i);
 elibera();}
void afisare()
{for (int i=1;i<=n;i++) cout<<c[i]<<" ";}
void main()
{citeste(); cout<<"nod de pornire: "; cin>>k; init(k);
 while (!este_vida()) prelucrare();
 cout<<"Nodurile vizitate prin metoda BF sunt: "<<endl; afisare();}

```

Atenție

Prin parcurgerea în lățime a unui graf, determinați lanțurile, respectiv drumurile de lungime minimă.

Complexitatea algoritmului de parcurgere în lățime

Algoritmul constă în eliminarea unui nod din coada de aşteptare și adăugarea vecinilor (succesorilor) săi în coada de aşteptare. Fiecare nod va fi adăugat în coada de aşteptare o dată și, prin urmare, va fi eliminat din coada de aşteptare o singură dată. Complexitatea operațiilor de eliminare a celor n noduri din coada de aşteptare este $O(n)$. Pentru a adăuga noduri la coada de aşteptare, sunt examinați vecinii (succesorii) nodului curent. Vecinii (succesorii) unui nod sunt examinați o singură dată, atunci când nodul este eliminat din coada de aşteptare. Numărul total de vecini examinați este egal cu m – numărul de muchii (arce) ale grafului. Complexitatea operațiilor de adăugare de noduri este $O(m)$. Complexitatea algoritmului este liniară $O(m+n)$.

Determinarea drumurilor de lungime minimă, între oricare două noduri din graf, cu ajutorul algoritmului de parcurgere în lățime

Algoritmul. Pentru a găsi drumurile de lungime minimă care pornesc din fiecare nod, se va parcurge graful în lățime de n ori (parcurgerea se va face pornind din fiecare dintre cele n noduri ale grafului). Pentru a putea reconstitui drumul, se folosește un vector (p) – pentru a memora predecesorul unui nod în parcurgerea în lățime:

Implementarea algoritmului. Se citește, din fișierul *graf21.txt*, matricea de adiacență a grafului neorientat G_{26} . Se afișează drumurile de lungime minimă dintre noduri. Se folosește variabila globală x , pentru nodul din care pornește drumul, și implicit, parcurgerea în lățime a grafului. Funcția *zero()* se folosește pentru a inițializa cu valoarea 0 vectorii *vizitat* și *p* înainte de a începe o nouă parcurgere a grafului, pornind din nodul x . Funcția *adaug()* a fost modificată prin adăugarea instrucțiunii de actualizare a vectorului *p* la adăugarea unui nod în coada de aşteptare (în elementul corespunzător nodului vizitat i se memorează predecesorul său în parcurgerea în lățime – nodul k). Funcția *afiseaza()* se folosește pentru a afișa drumurile de lungime minimă de la nodul x până la fiecare dintre celelalte noduri ale grafului. În această funcție se folosesc următoarele variabile locale: i – pentru a parcurge coada de aşteptare și vectorul *d*; j – pentru a parcurge nodurile grafului (aceste noduri sunt nodurile destinație ale drumurilor care pornesc din nodul x); *d* – un vector în care se memorează nodurile drumului (nodurile sunt înregistrate în ordine inversă):

de la predecesorul nodului j până la nodul x); y – pentru predecesorul unui nod și este – o variabilă de tip logic, pentru a verifica dacă există drum de la nodul x la nodul j .

nod	Vectorul p								
	1	2	3	4	5	6	7	8	9
p	0	1	1	1	2	3	4	6	6

```
#include <iostream.h>
typedef stiva[100];
int a[10][10], vizitat[10], c[10], p[10], n, prim, ultim, k, x;
fstream f("graf21.txt", ios::in);
void citeste() { //se citește matricea de adiacență din fișier}
void zero() {for (int i=1; i<=n; i++) {vizitat[i]=0; p[i]=0;}}
void init(int k) {prim=ultim=1; c[ultim]=k; vizitat[k]=1; p[k]=0;}
int este_vida() {return ultim<prim;}
void adaug(int i, int k) {ultim++; c[ultim]=i; vizitat[i]=1; p[i]=k;}
void prelucrare()
{int i; k=c[prim];
 for (i=1; i<=n; i++) if (a[k][i]==1 && vizitat[i]==0) adaug(i, k);
 prim++;}
void afisare()
{int i, j, este, y, d[10];
 cout<<"Drumul minim care porneste din nodul "<<x<<": "<<endl;
 for (j=1; j<=n; j++)
 if (j!=x)
 {cout<<"pana la nodul "<<j<<": ";
 for (i=1, este=0; i<=ultim && !este; i++) if (c[i]==j) este=1;
 if (este==1) {y=p[j]; i=0;
 while (y) {i++; d[i]=y; y=p[y];}
 for (i>1; i--) cout<<d[i]<<" "<<j<< " ";
 else cout<<"nu exista drum";
 cout<<endl;}
 cout<<endl;}
 cout<<endl;}
void main()
{citeste();
 cout<<"Drumurile de lungime minima intre doua noduri sunt: "<<endl;
 for (x=1; x<=n; x++)
 {zero(); init(x);
 while (!este_vida()) prelucrare();
 afisare();}}
```

2.7.8.2. Parcurgerea în adâncime – Depth First

Pentru această metodă de parcurgere a unui graf cu n noduri se folosesc următoarele structuri de date și date elementare:

1. **Variabilele de memorie.** n pentru numărul de noduri ale grafului, k pentru nodul care se prelucreză, i și j pentru parcurgerea matricei de adiacență și a vectorilor.
2. **Matricea de adiacență a grafului** a .
3. **Vectorul de vizitare** **vizitat**. El conține n elemente în care se înregistrează nodurile vizitate. Elementele vectorului **vizitat[i]** sunt definite ca și în metoda precedentă de parcurgere.
4. **Stiva** **st** a nodurilor parcurse. Stiva **st** gestionează nodurile vecine parcurse în adâncime. Pentru gestionarea stivei se folosește variabila de memorie **vf** pentru a identifica ultimul nod intrat în stivă. Pentru stivă se folosește implementarea statică cu

un vector st cu n elemente. În stivă vor fi înregistrate, în ordine, nodurile vizitate în adâncime, pe o direcție. Inițial, stiva conține un singur element care corespunde nodului cu care începe parcurgerea grafului, iar valoarea variabilei vf este 1. Pe măsură ce se parcurge graful, se completează următoarele elemente ale vectorului st . Atunci când se prelucrează un nod k , pe la vârful stivei se introduc în stivă toate nodurile i vecine cu nodul k care nu au fost vizitate încă.

Algoritmul pentru parcurgerea grafului este următorul:

- PAS1.** Se citesc din fișier valoarea pentru n și matricea de adiacență a grafului.
- PAS2.** Se initializează cu 0 elementele vectorului de vizitare $vizitat$, deoarece nici unul dintre noduri nu a fost vizitat.
- PAS3.** Se introduce de la tastatură valoarea variabilei de memorie k , corespunzătoare primului nod cu care începe parcurgerea grafului, și se afișează eticheta lui.
- PAS4.** Se initializează stiva st (vârful și conținutul vârfului stivei), astfel: $vf \leftarrow 1$; $st[vf] \leftarrow k$; deoarece inițial în stiva st este înregistrat doar nodul k cu care începe parcurgerea.
- PAS5.** Se actualizează vectorul $vizitat$ atribuind elementului k al vectorului valoarea 1, deoarece a fost vizitat ($vizitat[k] \leftarrow 1$).
- PAS6.** Cât timp stiva nu este vidă ($vf < > 0$) execută
 - PAS7.** Se extrage din vârful stivei, elementul corespunzător nodului căruia i se vor vizita vecinii ($k \leftarrow st[vf]$);.
 - PAS8.** Se initializează nodul i cu care începe căutarea ($i \leftarrow 1$);.
 - PAS9.** Cât timp nu s-a găsit un nod i vecin nodului k , nevizitat încă ($i <= n$ și $(a[i][k] = 0$ sau $(a[i][k] = 1$ și $vizitat[i] = 1$)) execută
 - PAS10.** Se trece la următorul nod, în vederea verificării ($i \leftarrow i + 1$), și se revine la Pasul 9.
- PAS11.** Dacă nu s-a mai găsit un nod i vecin nodului k nevizitat încă, atunci se elimină nodul k din stivă, prin coborârea vârfului stivei ($vf \leftarrow vf - 1$), altfel se afișează nodul găsit i, se adaugă în vârful stivei ($vf \leftarrow vf + 1$; $st[vf] \leftarrow i$); și se actualizează elementul din vectorul de vizitare care corespunde acestui nod, deoarece prin adăugarea nodului i la stiva st se consideră că acest nod a fost vizitat ($vizitat[i] \leftarrow 1$) și se revine la Pasul 6.

Pentru graful G_{26} , pornind din nodul 1, nodurile în stivă sunt adăugate astfel:

vizitat	1	0	0	0	0	0	0	0	0	
st	1									10

Inițializarea stivei cu primul nod: $vf = 1$.

vizitat	1	1	0	0	0	0	0	0		
st	1	2								10

Prelucrarea nodului: $vf = 1 \Rightarrow vf = 2$.

vizitat	1	1	0	0	1	0	0	0		
st	1	2	5							10

Prelucrarea nodului: $vf = 2 \Rightarrow vf = 3$.

vizitat	1	1	1	0	1	0	0	0		
st	1	2	3	4	5	6	7	8	9	10

Prelucrarea nodului: $vf = 3 \Rightarrow vf = 4$.

vizitat	1	1	1	0	1	1	0	0	0	
	1	2	3	4	5	6	7	8	9	10
st	1	2	5	3	6					

Prelucrarea nodului: vf=4 \Rightarrow vf=5.

vizitat	1	1	1	0	1	1	0	1	0	
	1	2	3	4	5	6	7	8	9	10
st	1	2	5	3	6	8				

Prelucrarea nodului: vf=5 \Rightarrow vf=6.

vizitat	1	1	1	0	1	1	0	1	0	
	1	2	3	4	5	6	7	8	9	10
st	1	2	5	3	6	8				

Prelucrarea nodului: vf=6 \Rightarrow vf=5.

vizitat	1	1	1	0	1	1	0	1	1	
	1	2	3	4	5	6	7	8	9	10
st	1	2	5	3	6	9				

Prelucrarea nodului: vf=5 \Rightarrow vf=6.

vizitat	1	1	1	0	1	1	0	1	1	
	1	2	3	4	5	6	7	8	9	10
st	1	2	5	3	6	9				

Prelucrarea nodului: vf=6 \Rightarrow vf=5.

vizitat	1	1	1	0	1	1	0	1	1	
	1	2	3	4	5	6	7	8	9	10
st	1	2	5	3	6					

Prelucrarea nodului: vf=5 \Rightarrow vf=4.

vizitat	1	1	1	0	1	1	1	1	1	
	1	2	3	4	5	6	7	8	9	10
st	1	2	5	3	7					

Prelucrarea nodului: vf=4 \Rightarrow vf=5.

vizitat	1	1	1	1	1	1	1	1	1	
	1	2	3	4	5	6	7	8	9	10
st	1	2	5	3	7	4				

Prelucrarea nodului: vf=6 \Rightarrow vf=5.

vizitat	1	1	1	1	1	1	1	1	1	
	1	2	3	4	5	6	7	8	9	10
st	1	2	5	3	7					

Prelucrarea nodului: vf=5 \Rightarrow vf=4.

vizitat	1	1	1	1	1	1	1	1	1	
	1	2	3	4	5	6	7	8	9	10
st	1	2	5	3						

Prelucrarea nodului: vf=4 \Rightarrow vf=3.

vizitat	1	1	1	1	1	1	1	1	1	
	1	2	3	4	5	6	7	8	9	10
st	1	2	5							

Prelucrarea nodului: vf=3 \Rightarrow vf=2.

vizitat	1	1	1	1	1	1	1	1	1	
	1	2	3	4	5	6	7	8	9	10
st	1	2								

Prelucrarea nodului: $vf=2 \Rightarrow vf=1$.

vizitat	1	1	1	1	1	1	1	1		
	1	2	3	4	5	6	7	8	9	10
st	1									

Prelucrarea nodului: $vf=1 \Rightarrow vf=0$. Terminare: $vf=0$

Pentru implementarea algoritmului se folosesc subprogramele:

- funcția procedurală `citeste` creează matricea de adiacență prin preluarea informațiilor din fișierul text `f`;
- funcția procedurală `init` initializează stiva cu primul nod vizitat;
- funcția operand `este_vida` testează stiva dacă este vidă;
- funcția procedurală `adauga` adaugă un nod la stivă;
- funcția procedurală `elimina` elimină nodul din vârful stivei;
- funcția procedurală `prelucrare` prelucrează nodul din vârful stivei: caută primul vecin nevizitat și dacă găsește un astfel de nod, îl afișează și îl adaugă în stivă; altfel, nodul din vârful stivei este eliminat (nu mai are vecini nevizitați);

Pentru testarea programului se folosește graful neorientat G_{26} a cărui matrice de adiacență se citește din fișierul text `graf21.txt`.

Observație. Pentru grafurile orientate, în funcția `prelucrare()` condiția structurii repetitive while este: `i <= n && ((a[k][i] == 0 || a[i][k] == 0) || ((a[k][i] == 1 || a[i][k] == 1) && vizitat[i] == 1))`

```
int n,a[10][10],vizitat[20],st[20],vf,k;
fstream f("graf21.txt",ios::in);
void citeste() { //se citește matricea de adiacență din fișier}
void init (int k) {vf=1; st[vf]=k; vizitat[k]=1;}
int este_vida() {return vf==0;}
void adauga(int i) {vf++; st[vf]=i; vizitat[i]=1;}
void elimina() {vf--;}
void prelucrare()
{int i=1; k=st[vf];
 while (i<=n && (a[k][i]==0 || (a[k][i]==1 && vizitat[i]==1))) i++;
 if (i==n+1) elimina(); else {cout<<i<<" "; adauga(i);}
}
void main()
{citeste(); cout<<"nodul de pornire: "; cin>>k;
 cout<<"Nodurile vizitate prin metoda DF sunt: "<<endl;
 cout<<k<<" "; init(k);
 while (!este_vida()) prelucrare();}
```

Atenție

Prin parcurgerea în adâncime a unui graf determinăm nodurile care aparțin aceleiași componente conexe, respectiv subrafului de succesiuni ai unui nod.

Complexitatea algoritmului de parcurgere în adâncime

Algoritmul constă în extragerea unui nod din stivă și adăugarea vecinilor (succesorilor) săi în stivă, iar dacă nu mai are vecini, nodul este eliminat din stivă. Fiecare nod va fi adăugat în stivă o dată și, prin urmare, va fi eliminat din stivă o singură dată. Complexitatea

operațiilor de eliminare a celor n noduri din stivă este $O(n)$. Pentru a adăuga noduri în stivă sunt examinați vecinii (succesorii) nodului din vârful stivei. Vecinii (succesorii) unui nod sunt examinați o singură dată, atunci când nodul este extras din stivă. Numărul total de vecini examinați este egal cu m – numărul de muchii (arce) ale grafului. Complexitatea operațiilor de adăugare de noduri este $O(m)$. Complexitatea algoritmului este liniară $O(m+n)$.

Determinarea conexității grafurilor cu ajutorul algoritmului de parcurgere în adâncime

1. Afisarea componentelor conexe dintr-un graf.

Algoritm. Identificarea multșimii de noduri care formează o componentă conexă se face parcurgând în adâncime graful pornind de la un nod inițial. Nodul cu care începe parcurgerea în adâncime pentru o componentă conexă se caută printre nodurile nevizitate încă. Graful se va parcurge în adâncime până când vor fi vizitate toate nodurile. Dacă graful este conex, se va afișa o singură componentă conexă – care va fi formată din toate nodurile grafului.

Implementarea algoritmului. Se citește din fișierul *graf21.txt* matricea de adiacență a a grafului neorientat G_{26} și se afișează toate componentele conexe ale grafului. Pe lângă variabilele și structurile de date folosite de algoritmul de parcurgere în adâncime se mai folosește variabila globală m , pentru a număra componentele conexe. Pentru a găsi nodul cu care se initializează parcurgerea se folosește funcția *cauta()* care cauță primul nod nevizitat, în ordinea etichetelor nodurilor.

```
int n,a[10][10],vizitat[20],st[20],vf,k,m;
fstream f("graf21.txt",ios::in);
void citeste() {//se citește matricea de adiacență din fișier}
int cauta()
{
    for (int i=1; i<=n; i++) if (vizitat[i]==0) return i;
    return 0;
}
void init(int k) {vf=1; st[vf]=k; vizitat[k]=1;}
int este_vida() {return vf==0;}
void adaug(int i) {vf++; st[vf]=i; vizitat[i]=1;}
void elimin() {vf--;}
void prelucrare()
{
    int i=1; k=st[vf];
    while (i<=n && (a[k][i]==0 || (a[i][k]==1 && vizitat[i]==1))) i++;
    if (i==n+1) elimin(); else {cout<<i<<" "; adaug(i);}
}
void main()
{citeste(); k=cauta();
while (k)
{m++; init(k); cout<<endl<<"Componentă conexă "<<n<<": "<<k<< " ";
 while (!este_vida()) prelucrare();
 k=cauta();}}
```

2. Afisarea componentelor tare conexe dintr-un graf orientat.

Algoritm. Nodul cu care începe parcurgerea în adâncime pentru o componentă tare conexă se caută printre nodurile nevizitate încă. Pentru acest nod se determină multimea nodurilor care formează subgraful succesorilor și multimea nodurilor care formează subgraful predecesorilor. Prin intersecția celor două subgrafuri (multimi de noduri) se obține componenta tare conexă. Identificarea celor două multimi de noduri se face parcurgând în adâncime graful pentru fiecare multime, pornind de la un nod inițial. Un nod se consideră

vizitat numai dacă a fost adăugat la o componentă tare conexă. Prelucrarea componentelor tare conexe prin parcurgerea în adâncime a grafului se va face până când vor fi vizitate toate nodurile. Dacă graful este tare conex, se va afișa o singură componentă tare conexă care va fi formată din toate nodurile grafului.

Implementarea algoritmului. Se citește din fișierul *graf20.txt* matricea de adiacență a grafului orientat G_{22} și se afișează toate componentele tare conexe ale grafului. Vectorii *pred* și *succ* se folosesc pentru a determina subgraful predecesorilor, respectiv subgraful succesorilor nodului care se prelucrează. Vectorii au n elemente. Fiecare element i are valoarea 1 dacă nodul i aparține subgrafului predecesorilor, respectiv subgrafului succesorilor; altfel, are valoarea 0. Funcția *zero()* se folosește pentru a inițializa cu valoarea 0 vectorii *pred* și *succ* înainte de prelucrarea unei componente tare conexe. Pentru a găsi nodul cu care se inițializează o componentă tare conexă, se folosește funcția *cauta()* care caută primul nod nevizitat, în ordinea etichetelor nodurilor. Funcțiile *prelucrare1()* și *prelucrare2()* se folosesc pentru a parcurge graful în adâncime în vederea determinării subgrafului predecesorilor, respectiv subgrafului succesorilor nodului care se prelucrează. Funcția *comp()* se folosește pentru a determina nodurile unei componente tare conexe prin intersecția vectorilor *pred* și *succ*.

```

int n,a[10][10],vizitat[20],st[20],vf, m;
fstream f("graf20.txt",ios::in);
void citeste(){//se citește matricea de adiacență din fișier}
int cauta() { //este identică cu cea de la exemplul anterior}
void zero(int x[]) {for(int i=1;i<=n;i++) x[i]=0;}
void init(int k) {vf=1; st[vf]=k;}
int este_vida() {return vf==0;}
void adaug(int i,int x[]) {vf++; st[vf]=i; x[i]=1;}
void elimin() {vf--;}
void prelucrare1(int x[])
{int i=1,k=st[vf]; x[k]=1;
 while (i<=n && (a[i][k]==0 || (a[i][k]==1 & x[i]==1))) i++;
 if (i==n+1) elimin(); else adaug(i,x);}
void prelucrare2(int x[])
{int i=1,k=st[vf]; x[k]=1;
 while (i<=n && (a[k][i]==0 || (a[k][i]==1 & x[i]==1))) i++;
 if (i==n+1) elimin(); else adaug(i,x);}
void comp(int x[], int y[])
{for(int i=1;i<=n;i++)
 if(x[i]==1 && y[i]==1) {cout<<i<<" "; vizitat[i]=1;}}
void main()
{int k,pred[10],succ[10];
 citeste(); k=cauta(); m++; cout<<"componentele tare conexe: "<<endl;
 while (k)
 {cout<<endl<<m<<": "; init(k); zero(pred);
 while (!este_vida()) prelucrare1(pred);
 init(k); zero(succ);
 while (!este_vida()) prelucrare2(succ);
 comp(pred,succ); k=cauta(); m++;}}

```



1. Scrieți un program care citește din fișierul text *graf4.txt* lista arcelor unui graf orientat și care, folosind algoritmul corespunzător de parcursare a unui graf orientat, realizează:
 - a. Afisează lanțurile de lungime minimă dintre noduri.
 - b. Verifică dacă două noduri x și y aparțin aceleiași componente conexe (și varianta, apărând în aceleiași componente tare conexe). Etichetele nodurilor se citesc de la tastatură.
 - c. Verifică dacă graful este tare conex și afișează un mesaj de informare.
2. În nodurile unui graf se păstrează numere naturale. Graful pune în evidență divizorii unui număr, astfel: numărul x memorat în nodul i este în relație cu numărul y memorat în nodul j , dacă numărul y este divizor al numărului x . Să se verifice dacă în graf se găsesc toți divizorii unui număr p citit de la tastatură. (Indicație. Se parcurge mai întâi graful, pornind de la nodul cu eticheta 1 – pentru a găsi numărul – și, în cazul în care se găsește, se reia parcurgerea grafului pornind cu nodul în care s-a găsit numărul – și se verifică dacă au fost găsiți toți divizorii. Pentru testarea programului se va crea un graf al divizorilor care conține următoarele numere: $\{2,3,4,5,6,9,12,15,18,20,38\}$ și se vor căuta divizorii numărului 36 și apoi ai numărului 20.)

2.7.8.3. Aplicații practice

1. Din grupul de n persoane între care s-au stabilit relații de prietenie, afișați cel mai mare grup de prieteni. (Indicație. Se determină componenta conexă care conține cele mai multe noduri).
2. Din grupul de n persoane între care s-au stabilit relații de cunoștință, afișați grupul format din cele mai multe persoane care se cunosc reciproc. (Indicație. Se determină componenta tare conexă care conține cele mai multe noduri.)
3. Din grafurile care pun în evidență relațiile de vecinătate și de prietenie ale sătenilor care au fânețe, obțineți următoarele informații:
 - a. Care sunt sătenii cei mai îndepărtați de drumul sătesc. (Indicație. Găsiți cel mai lung lanț din lanțurile de lungime minimă din graful fânețelor.)
 - b. Care sunt sătenii izolați. (Indicație. Sătenii care nu se găsesc în graful de vecinătate într-o componentă conexă în care se află și un sătean cu ieșire la drumul sătesc.)
4. Din grafurile care pun în evidență șosele din zona turistică, stabiliți căte localități rămân izolate dacă se distrug un pod pe care trece șoseaua care leagă două localități ale căror etichete se citesc de la tastatură.
5. În rețeaua de străzi a orașului, stabiliți cel mai scurt drum între două intersecții ale căror etichete se citesc de la tastatură.

2.7.9. Graful ponderat

2.7.9.1. Definiția grafului ponderat

Considerăm un graf $G=(X,U)$ și o funcție $f:U \rightarrow R_+$ care asociază fiecărei muchii (arc) u un număr real pozitiv (care poate avea semnificația de cost, distanță, timp, durată), numită în general **costul muchiei**. Funcția f se numește **funcția cost**.

Un graf $G = (X,U)$ pentru care s-a definit o funcție cost se numește graf ponderat.

Observații:

1. Graful ponderat se mai numește și **graf cu costuri**.
2. Grafurile ponderate se folosesc în aplicări în care trebuie determinată valoarea minimă sau valoarea maximă a unei mărimi asociate grafului, adică a funcției cost.

3. Se definește **costul unui drum** de la nodul x la nodul y ca fiind suma costurilor muchiilor (arcelor) care formează acel drum.
4. Metoda cea mai adekvată pentru reprezentarea unui graf ponderat este **matricea costurilor**.

Exemplu – Graful G_{27} din figura 32.

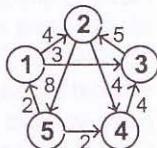


Fig. 32

Studiu de caz

Scop: exemplificarea unei aplicații în care pentru reprezentarea datelor se folosește un graf ponderat.

Exemplul 1. O firmă detine depozite de marfă în mai multe localități. O rețea de șosele leagă direct unele dintre aceste localități. Distanța dintre două locații este măsurată în kilometri. Să se determine traseul pe care trebuie să-l parcurgă o mașină pentru a transporta marfă de la depozitul din orașul A la depozitul din orașul B astfel încât distanța parcursă în kilometri să fie minimă.

Localitățile formează nodurile unui graf neorientat în care fiecare muchie reprezintă o legătură directă pe șosea între două localități. Fiecare muchie are asociată o mărime – distanță. Această mărime este costul muchiei, iar graful este un graf ponderat. Cerința problemei este de a determina un drum cu lungime minimă (cu costul minim) între două noduri ale grafului.

Exemplul 2. O persoană trebuie să se deplaseze cu autoturismul cât mai repede între două intersecții din oraș. Traficul între două intersecții nu este întotdeauna în ambele sensuri. Cunoscând timpul mediu de deplasare între două intersecții, să se determine care este traseul pe care trebuie să-l aleagă pentru a ajunge de la intersecția A la intersecția B cât mai repede.

Intersecțiile formează nodurile unui graf orientat în care fiecare arc reprezintă sensul de circulație pe o stradă care leagă direct două intersecții. Fiecare arc are asociată o mărime – timpul mediu de parcurgere. Această mărime este costul muchiei, iar graful este un graf ponderat. Cerința problemei este de a determina un drum cu timpul de parcurgere minim (cu costul minim) între două noduri ale grafului.

Observație. Orice problemă la care soluția este de a găsi drumul cu costul minim (sau maxim) dintre două puncte se rezolvă reprezentând datele printr-un graf ponderat și folosind un algoritm de determinare a drumului cu costul minim (respectiv maxim) dintre două noduri ale grafului.



2.7.9.2. Matricea costurilor

Matricea costurilor unui **graf** este o matrice pătratică de dimensiune n ($A_{n,n}$), ale cărei elemente $a_{i,j}$ sunt definite astfel încât să pună în evidență costul asociat fiecărei muchii (arc).

În funcție de cerința aplicației, există două forme de reprezentare a matricei costurilor:

- **matricea costurilor minime** – pentru a determina valoarea minimă a funcției cost;
- **matricea costurilor maxime** – pentru a determina valoarea maximă a funcției cost.

a) **Matricea costurilor minime.** Elementele $a_{i,j}$ ale matricei sunt definite astfel:

$$a_{i,j} = \begin{cases} c, & \text{dacă există o muchie (un arc) cu costul } c > 0 \text{ între nodurile } i \text{ și } j, \text{ cu } i \neq j \\ 0, & \text{dacă } i = j \\ \infty, & \text{dacă nu există o muchie (un arc) între nodurile } i \text{ și } j, \text{ cu } i \neq j \end{cases}$$

Fiecarei muchii (arc) care nu există în graf i se asociază o valoare foarte mare, deoarece, căutându-se costul minim, această muchie (arc) nu va mai fi selectată. Deoarece pentru implementarea matricei nu se poate folosi simbolul ∞ , în locul lui se va folosi cea mai mare valoare ce se poate reprezenta în calculator pentru tipul de dată asociat costului.

Exemplu. Pentru graful ponderat G_{27} din figura 32 matricea costurilor minime este prezentată alăturat.

	1	2	3	4	5
1	0	4	3	∞	∞
2	∞	0	∞	7	8
3	∞	∞	0	∞	∞
4	∞	∞	4	0	∞
5	2	∞	∞	2	0

b) **Matricea costurilor maxime.** Elementele $a_{i,j}$ ale matricei sunt definite astfel:

$$a_{i,j} = \begin{cases} c, & \text{dacă există o muchie (un arc) cu costul } c > 0 \text{ între nodurile } i \text{ și } j, \text{ cu } i \neq j \\ 0, & \text{dacă } i = j \\ -\infty, & \text{dacă nu există o muchie (un arc) între nodurile } i \text{ și } j, \text{ cu } i \neq j \end{cases}$$

Fiecarei muchii (arc) care nu există în graf i se asociază o valoare foarte mică, deoarece, căutându-se costul maxim, această muchie (arc) nu va mai fi selectată. Deoarece pentru implementarea matricei nu se poate folosi simbolul $-\infty$, în locul lui se va folosi cea mai mică valoare ce se poate reprezenta în calculator pentru tipul de dată asociat costului.



Tema

Scriți matricea costurilor maxime pentru graful din figura 32.

Algoritmul pentru crearea matricei costurilor

Pentru a crea matricea costurilor trebuie să se citească pentru fiecare muchie (arc) nodurile de la extremități și costul asociat fiecarei muchii (arc). Aceste informații se pot citi de la tastatură sau dintr-un fișier. Algoritmul pentru crearea matricei costurilor este:

- PAS1.** Se inițializează matricea astfel: toate elementele de pe diagonala principală cu valoarea 0, iar restul elementelor cu valoarea corespunzătoare pentru ∞ ($-\infty$).
- PAS2.** Se actualizează matricea cu informațiile despre costurile asociate muchiilor (arcelor) astfel: pentru fiecare muchie (arc) $[i,j]$ cu costul c , elementului $a[i][j]$ i se va atribui valoarea costului c .

Implementarea algoritmului pentru matricea costurilor minime a unui graf orientat. Pentru inițializarea matricei costurilor se folosește funcția `init()`, iar pentru actualizarea ei funcția `citire()`. Elementele matricei fiind de tip `int` nu se poate folosi pentru simbolul ∞ constanta de sistem `MAXINT`, deoarece în algoritmii de determinare a drumului cu costul minim prin adunările repetate ale elementelor matricei (care pot avea și valoarea `MAXINT`) se depășește capacitatea de reprezentare a tipului `int`. Există două soluții:

- a. pentru elementele matricei se alege tipul `long`, chiar dacă acest tip de dată nu este justificat de valorile foarte mici ale costurilor (și se obține o utilizare ineficientă a memoriei interne);
- b. se definește o constantă cu o valoare foarte mare în comparație cu celelalte costuri.

În implementarea algoritmului s-a ales varianta unei constante definite – `MAX`. Datele se citesc din fișierul text `cost.txt` în care pe prima linie există un număr care reprezintă numărul de noduri ale grafului, iar pe următoarele rânduri câte trei valori numerice separate prin spațiu, care reprezintă nodurile de la extremitatea unui arc – i și j – și costul asociat arcului – c . Pentru testarea programului se folosește graful G_{27} .

```
#include <iostream.h>
int a[100][100],n;
int const MAX=5000;
fstream f("cost.txt",ios::in);
void init() //se initializeaza matricea costurilor
{int i,j; f>>n;
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++) if (i==j) a[i][j]=0;
    else a[i][j]=MAX;}
void citire() //se actualizeaza matricea costurilor cu datele din fisier
{int i,j,c;
while (f>>i>>j>>c) a[i][j]=c; f.close();}
void main() {...}
```

Temă

Scripteți un program care creează matricea costurilor maxime pentru graf neorientat. Pentru testare considerați în graful din figura 32 că arcele sunt muchii.

2.7.9.3. Algoritmi pentru determinarea costului minim (maxim)

Pentru determinarea drumului cu costul minim (maxim) între două noduri ale unui graf se poate folosi:

- **algoritmul Roy-Floyd;**
- **algoritmul Dijkstra.**

Ambii algoritmi folosesc principiul enunțat prin teorema lui Bellman: **drumul optim** (cu costul minim, respectiv maxim) între două noduri oarecare i și j conține numai drumuri parțiale optime (cu costuri minime, respectiv maxime) care trec prin alte noduri ale grafului. Altfel spus, dacă drumul optim dintre două noduri oarecare i și j trece prin un nod k , atunci și drumurile de la i la k și de la k la j sunt optime. Cei doi algoritmi diferă prin modul în care se identifică nodurile intermedii k .

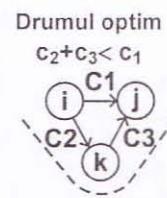


Fig. 33

a) Algoritmul Roy-Floyd

Algoritmul folosește un principiu asemănător cu cel care a fost utilizat pentru determinarea matricei drumurilor: găsirea drumului optim între două noduri oarecare i și j prin descoperirea drumurilor optime care îl compun și care trec prin nodurile k se face prin transformarea matricei costurilor. Matricea trece prin n transformări, în urma cărora fiecare element $a[i][j]$ va memora costul drumului minim dintre nodurile i și j .

PAS1. Pentru etichete ale nodului k de la 1 la n (adică pentru orice nod al grafului) execută:

PAS2. Pentru orice pereche de noduri din graf i și j (cu $1 \leq i \leq n$ și $1 \leq j \leq n$) execută:

PAS3. Dacă suma dintre costul drumului de la i la k și costul drumului de la k la j ($a[i][k] + a[k][j]$) este mai mică decât costul drumului de la i la j ($a[i][j]$), atunci în matricea costurilor costul drumului direct de la i la j este înlocuit cu costul drumului care trece prin nodul k ($a[i][j] = a[i][k] + a[k][j]$).

Pentru graful din figura 32 matricea costurilor suferă următoarele cinci transformări. La fiecare transformare, dacă drumul de la nodul i la nodul j are costul mai mare decât costul drumurilor care trec prin nodul intermedian k (de la nodul i la nodul k și de la nodul k la nodul j), atunci elementului $a[i][j]$ îi se va atribui valoarea $a[i][k] + a[k][j]$.

k=1					
1	2	3	4	5	
1	0	4	3	∞	∞
2	∞	0	∞	7	8
3	∞	5	0	∞	∞
4	∞	∞	4	0	∞
5	2	6	5	2	0

k=2					
1	2	3	4	5	
1	0	4	3	11	12
2	∞	0	∞	7	8
3	∞	5	0	12	13
4	∞	∞	4	0	∞
5	2	6	5	2	0

k=3					
1	2	3	4	5	
1	0	4	3	11	12
2	∞	0	∞	7	8
3	∞	5	0	12	13
4	∞	9	4	0	17
5	2	6	5	2	0

k=4					
1	2	3	4	5	
1	0	4	3	11	12
2	∞	0	11	7	8
3	∞	5	0	12	13
4	∞	9	4	0	17
5	2	6	5	2	0

k=5					
1	2	3	4	5	
1	0	4	3	11	12
2	10	0	11	7	8
3	15	5	0	12	13
4	19	9	4	0	17
5	2	6	5	2	0

Interpretarea datelor din matricea obținută în urma transformărilor se face astfel: drumul de la nodul i la nodul j are costul minim $a[i][j]$. De exemplu, drumul cu costul minim de la nodul 2 la nodul 4 are costul minim 7. **Matricea nu furnizează informații despre etichetele drumului cu costul minim.**

Pentru **implementarea algoritmului** se folosesc subprogramele:

- funcția procedurală **init** initializează matricea costurilor;
- funcția procedurală **citire** actualizează matricea costurilor cu datele din fișier;
- funcția procedurală **transformare** transformă matricea costurilor;
- funcția procedurală **afisare** afișează lungimea drumurilor minime între toate nodurile grafului.

```
#include <fstream.h>
int a[100][100],n;
int const MAX=5000;
fstream f("cost.txt",ios::in);
void init() { //se initializeaza matricea costurilor}
void citire() { //se actualizeaza matricea costurilor cu datele din fisier}
void transformare() //se transforma matricea costurilor
{for(int k=1;k<=n;k++)
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
            if(a[i][k]+a[k][j]<a[i][j]) a[i][j]=a[i][k]+a[k][j];}
void afisare()
{cout<<"costul drumurilor minime intre nodurile: "<<endl;
 for(int i=1;i<=n;i++)
    for(int j=1;j<=n;j++)
        if(a[i][j]<MAX && i!=j) cout<< "("<<i<<","<<j<<") - "<<a[i][j]<<endl;}
void main()
{init(); citire(); transformare(); afisare();}
```

Informațiile din matricea costurilor transformată prin algoritmul Roy Floyd se pot folosi pentru a **verifica dacă există drum cu costul minim între două noduri ale grafului**, iar în caz afirmativ, se poate afișa lungimea lui și se poate descoperi drumul.

Algoritmul de descoperire a drumului cu costul minim pornind de la matricea costurilor transformată folosește același raționament ca la transformarea ei: dacă lungimea drumului minim dintre nodurile i și j este egală cu suma dintre lungimile minime a două drumuri care trec prin un nod intermediu k ($a[i][k]+a[k][j]=a[i][j]$), atunci nodul k face parte

din drumul de lungime minimă de la i la j . Deoarece problema pentru determinarea nodurilor care formează drumul de lungime minimă se descompune în două subprobleme: determinarea drumului minim de la nodul i la nodul k (cu $k \neq i$) și determinarea drumului minim de la nodul k la nodul j (cu $k \neq j$). În implementarea algoritmului se folosește strategia **divide et impera**.

Pentru **implementarea algoritmului** prin care se determină drumul minim dintre două noduri x și y (a căror etichetă se citește de la tastatură) se folosesc subprogramele:

- funcția procedurală **init** inițializează matricea costurilor;
- funcția procedurală **citire** actualizează matricea costurilor cu datele din fișier;
- funcția procedurală **transformare** transformă matricea costurilor;
- funcția procedurală **drum** determină nodurile drumului cu cost minim;
- funcția procedurală **afisare** afișează costul drumului minim și nodurile care formează drumul;

```
#include <fstream.h>
int a[100][100],n;
int const MAX=5000;
fstream f("cost.txt",ios::in);
void init() { //se initializeaza matricea costurilor}
void citire() { //se actualizeaza matricea costurilor cu datele din fișier}
void transformare() { //se transforma matricea costurilor }
void drum(int i, int j) //se determină nodurile drumului minim
{int k,gasit;
for (k=1,gasit=0; k<=n && !gasit; k++)
    if (i!=k && j!=k) && (a[i][j]==a[i][k]+a[k][j])
        {drum(i,k); drum(k,j); gasit=1;}
if (!gasit) cout<<j<<" ";
void afisare(int x, int y)
{if (a[x][y]<MAX)
    {cout<<"drumul minim de la nodul "<<x<<" la nodul "<<y;
     cout<<" are costul "<<a[x][y]<<endl;
     cout<<x<<" "; drum(x,y);}
else cout<<"Nu exista drum";}
void main()
{int x,y; cout<<"x= "; cin>>x; cout<<"y= "; cin>>y;
init(); citire(); transformare(); afisare(x,y); }
```

Complexitatea algoritmului Roy-Floyd

Algoritmul de transformare a matricei costurilor are ordinul de complexitate $O(n \times n \times n) = O(n^3)$ deoarece fiecare structură repetitivă **for** se execută de n ori, iar structurile **for** sunt imbricate. Algoritmul de determinare a drumurilor cu costul minim din matricea costurilor transformată are ordinul de complexitate al algoritmului **divide et impera**: $O(n \times \lg_2 n)$. Ordinul algoritmului este $O(n^3) + O(n \times \lg_2 n) = O(n^3 + n \times \lg_2 n) = O(n^3)$.

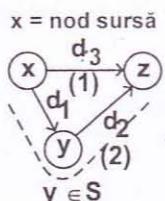
 Modificați programele care implementează algoritmul Roy Floyd astfel încât să se determine drumurile cu costul maxim.

b) Algoritmul Dijkstra

Algoritmul lui Dijkstra construiește drumurile cu costul minim care pornesc de la un nod oarecare x – nodul sursă – până la fiecare nod din graful $G=(X,U)$ – nodul destinație. Algoritmul întreține o mulțime cu nodurile care au fost deja selectate – S , și o **coadă de priorități** Q cu nodurile care nu au fost selectate încă: $Q=X-S$, astfel:

- Un nod y este declarat selectat atunci când s-a determinat costul final al drumului cu **costul minim** de la nodul sursă x la el. Selectarea unui nod nu este echivalentă cu găsirea drumului cu costul minim deoarece este posibil ca în urma calculării costului să rezulte că nu există drum de la nodul x la acel nod.
- În coada Q prioritatea cea mai mare o are nodul pentru care costul drumului are valoarea cea mai mică dintre toate costurile de drumuri care pornesc de la nodul x la celelalte noduri neselectate încă. La fiecare extragere a unui nod din coada de priorități Q , nodul este adăugat la mulțimea S , iar coada de priorități este reorganizată în funcție de acest nod (se recalculează costul drumurilor de la nodul x la nodurile rămase în coadă, considerând că unele drumuri, dacă trec și prin nodul extras, pot să-și micșoreze costul). Pentru calcularea drumurilor de lungime minimă se întreține o mulțime D în care se memorează costul drumurilor de la nodul x la nodurile neselectate, costuri care se recalculează la fiecare extragere de nod.

Drumul cu costul minim care pornește din nodul x este format din nodul inițial x și crește până când coada de priorități Q nu mai conține noduri. Deoarece, cele două mulțimi S și Q sunt disjuncte, iar reuniunea lor este mulțimea nodurilor X , este suficient să se întrețină numai mulțimea S . Algoritmul folosește strategia **greedy**, deoarece întotdeauna alege nodul cel mai apropiat de nodul sursă x .



$z \in Q$ (nod neselectat)
 (1) $d_3 < d_1 + d_2 : x \rightarrow y$
 (2) $d_3 > d_1 + d_2 : x \rightarrow y \rightarrow z$

Fig. 34

PAS1. Se initializează: $S = \emptyset$, se citește nodul inițial x și se atribuie mulțimii S .

PAS2. Se initializează mulțimea D cu costurile drumurilor de la nodul x la toate celelalte noduri ale grafului (sunt preluate din matricea costurilor elementele $a[x][i]$).

PAS3. Cât timp coada de priorități Q nu este vidă (mai există noduri neselectate) execută

PAS4. Se caută printre nodurile neselectate nodul y cu cel mai mic cost al drumului (reprezintă elementul care trebuie eliminat din coada de priorități Q).

PAS5. Se adaugă nodul y la mulțimea S : $S = S \cup \{y\}$ (înseamnă extragerea nodului y din coada de priorități Q și declararea lui ca nod selectat).

PAS6. Pentru fiecare nod neselectat (nod din coada de priorități) execută

PAS7. Se recalculează costul drumului de la nodul x la acest nod folosind ca nod intermediu nodul extras.

PAS8. Dacă acest cost este mai mic decât cel din mulțimea D , atunci el va fi noul cost.

Implementarea algoritmului. Se folosesc trei vectori:

→ Vectorul s pentru mulțimea nodurilor selectate, definit astfel:

$$s(i) = \begin{cases} 0, & \text{dacă nodul } i \text{ nu a fost selectat} \\ 1, & \text{dacă nodul } i \text{ a fost selectat} \end{cases}$$

Inițial, elementele vectorului s au valoarea 0, cu excepția elementului $s[x]$ care are valoarea 1. La terminarea execuției algoritmului, toate elementele din vectorul s vor avea valoarea 1. Nodurile i pentru care $s[i] = 0$ se consideră că fac parte din coada de priorități Q .

→ Vectorul d conține costul drumurilor, astfel: $d[i] =$ costul drumului minim găsit la un moment dat de la nodul x la nodul i (cu $1 \leq i \leq n$). Inițial $d[i] = a[x][i]$. La terminarea algoritmului, $d[i] =$ costul minim al drumului de la nodul x la nodul i .

→ Vectorul p memorează drumurile găsite între nodul x și celelalte noduri i ale grafului. Memorarea drumului se face prin legătura cu predecesorul care este definită astfel: $p[i]$ memorează nodul j care este predecesorul nodului i pe drumul de la x , cu

exceptia nodului sursă pentru care $p[x]=0$. Inițial, pentru toate nodurile i care nu au costul infinit (pentru care există un arc de la nodul x la nodul i), $p[i]=x$; altfel $p[i]=0$.

Nodul i care se extrage din coada de priorități Q trebuie să îndeplinească următoarele condiții:

$$\rightarrow s[i]=0.$$

$$\rightarrow d[i] = \min\{d[j] \mid 1 \leq j \leq n; s[j]=0\}.$$

$d[i]$ reprezintă costul minim al drumului de la nodul x la nodul i .

Pentru reorganizarea cozii de priorități se procedează astfel: pentru fiecare nod j cu $s[j]=0$ se calculează costul drumului de la nodul x la nodul j care trece prin nodul i : $d[i]+a[i][j]$. Dacă acest cost este mai mic decât $d[j]$, atunci aceasta va fi noua valoare a lui $d[j]$ și se actualizează vectorul p : $p[j]=i$.

Pentru graful din figura 32, considerând $x=1$, algoritmul se execuță astfel:

Inițial:

Vectorii	1	2	3	4	5
s	1	0	0	0	0
d	0	4	3	∞	∞
p	0	1	1	0	0

Drumul cu costul cel mai mic este cu nodul 3: $d[3]=3$. Nodul 3 se va extrage din coada Q.

Se analizează nodurile care rămân în coada de priorități:

Nodul 2. $d[3]+a[3][2] = 3+5 = 8 > 4$. Nu se modifică nimic.

Nodul 4. $d[3]+a[3][4] = 3+\infty = \infty > 4$. Nu se modifică nimic.

Nodul 5. $d[3]+a[3][5] = 3+\infty = \infty > 4$. Nu se modifică nimic.

Vectorii	1	2	3	4	5
s	1	0	1	0	0
d	0	4	3	∞	∞
p	0	1	1	0	0

Drumul cu costul cel mai mic este cu nodul 2: $d[2]=4$. Nodul 2 se va extrage din coada Q.

Se analizează nodurile care rămân în coada de priorități:

Nodul 4. $d[2]+a[2][4] = 4+7 = 11 < \infty$. Se modifică: $d[4]=11$ și $p[4]=2$.

Nodul 5. $d[2]+a[2][5] = 4+8 = 12 < \infty$. Se modifică: $d[5]=12$ și $p[5]=2$.

Vectorii	1	2	3	4	5
s	1	1	1	0	0
d	0	4	3	11	12
p	0	1	1	2	2

Drumul cu costul cel mai mic este cu nodul 4: $d[4]=11$. Nodul 4 se va extrage din coada Q.

Se analizează nodurile care rămân în coada de priorități:

Nodul 5. $d[4]+a[4][5] = 11+\infty = \infty > 12$. Nu se modifică nimic.

Vectorii	1	2	3	4	5
s	1	1	1	1	0
d	0	4	3	11	12
p	0	1	1	2	2

Drumul cu costul cel mai mic este cu nodul 5: $d[5]=15$. Nodul 5 se va extrage din coada Q.

Coada este vidă și se termină execuția algoritmului.

Final:

Vectorii	1	2	3	4	5
s	1	1	1	1	1
d	0	4	3	11	12
p	0	1	1	2	2

Din datele care se găsesc în vectorii d și p la terminarea algoritmului se obțin următoarele informații:

- $d[i]$ reprezintă costul minim al drumului de la nodul x la nodul i . De exemplu, pentru nodul 4 costul minim este 11.
- Din vectorul predecesorilor se reconstituie drumul cu costul minim de la nodul x la nodul i . De exemplu, pentru nodul 4: $p[4]=2$, iar $p[2]=1$. Drumul este $1 \rightarrow 2 \rightarrow 4$.

Pentru implementarea algoritmului în care se determină drumul cu costul minim dintre două noduri x și y (a căror etichetă se citește de la tastatură) se folosesc subprogramele:

- funcția procedurală **init** initializează matricea costurilor;
- funcția procedurală **citire** actualizează matricea costurilor cu datele din fișier;
- funcția procedurală **generare_drum** transformă vectorii d și p conform algoritmului pentru a obține drumurile cu costul minim de la nodul x la oricare alt nod i din graf;
- funcția procedurală **drum** determină nodurile drumului cu cost minim de la nodul x până la un nod i din graf – folosind informațiile din vectorul p ;
- funcția procedurală **afisare** afișează lungimea drumurilor minime care pornesc din nodul x până la fiecare nod i din graf și nodurile care formează drumul;

```
#include <iostream.h>
int a[100][100],d[100],s[100],p[100],n;
int const MAX=5000;
fstream f("cost.txt",ios::in);
void init() { //se initializeaza matricea costurilor}
void citire() { //se actualizeaza matricea costurilor cu datele din fisier}
void generare_drum(int x) //se genereaza drumurile
{int i,j,min,y; s[x]=1;
 for(i=1;i<=n;i++)
 {d[i]=a[x][i];
 if (i!=x && d[i]<MAX) p[i]=x;}
 for(i=1;i<=n-1;i++)
 {for(j=1,min=MAX; j<=n; j++)
 if (s[j]==0 && d[j]<min) {min=d[j]; y=j;}
 s[y]=1;
 for(j=1;j<=n;j++)
 if (s[j]==0 && d[j]>d[y]+a[y][j]) {d[j]=d[y]+a[y][j]; p[j]=y;}}
 void drum(int i)
 {if (p[i]!=0) drum(p[i]); cout<<i<<" ";}
 void afisare(int x)
 {for(int i=1;i<=n;i++)
 if (i!=x)
 if (p[i]!=0)
 {cout<<"drumul cu costul minim de la nodul "<<x;
 cout<<" la nodul "<<i<<" are costul "<<d[i]<<endl;
 drum(i); cout<<endl;}
 else cout<<"Nu exista drum de la "<<x<<" la "<<i<<endl;}
 void main()
 {int x; cout<<"x= "; cin>>x;
 init(); citire(); generare_drum(x); afisare(x);}
```

Complexitatea algoritmului lui Dijkstra

Pentru determinarea drumului cu costul minim se execută pașii algoritmului. Pasul 2 are ordinul de complexitate $O(n)$. Pasul 3 se execută pentru fiecare nod din graf, mai puțin nodul sursă, deoarece fiecare nod trebuie selectat o dată (se execută de $n-1$ ori). Pentru

fiecare nod selectat se analizează celelalte noduri, executându-se: Pasul 4 de n ori (se caută printre toate cele n noduri dacă mai există în coada de priorități, iar printre cele care mai sunt în coada de priorități se caută nodul cu costul drumului cel mai mic), iar Pasul 6 de n ori deoarece trebuie identificate printre cele n noduri care mai sunt în coada de priorități. Ordinul de complexitate al algoritmului pentru determinarea drumului cu costul minim va fi: $O(n) + O(nx(n+n)) = O(n) + O(n^2) = O(n^2)$. În algoritmul pentru afișarea drumului sunt analizate toate cele n noduri ale grafului, iar pentru fiecare nod i se determină recursiv drumul. Complexitatea algoritmului de afișare va fi $O(n) \times O(n \lg_2 n) = O(n^2 \lg_2 n)$.

 Temă

Modificați programul care implementează algoritmul Dijkstra astfel încât să se determine drumurile cu costul maxim.

2.7.9.4. Aplicații practice

1. O persoană oficială trebuie să se deplaseze într-un oraș între două puncte situate fiecare într-o intersecție. Traficul între două intersecții nu este întotdeauna în ambele sensuri, dar între oricare două intersecții există trafic prin intermediul altor intersecții. Pe fiecare stradă a traseului trebuie să existe un anumit număr de agenți care să asigure securitatea persoanei oficiale. Scrieți un program care să determine traseul astfel încât numărul de agenți să fie minim. Datele se citesc dintr-un fișier, astfel: de pe primul rând, numărul de intersecții, iar de pe următoarele rânduri, triplete de numere x, y, z care semnifică faptul că pentru traficul de la intersecția x până la intersecția y sunt necesari z agenți (cele două intersecții sunt legate direct).
2. O firmă are mai multe puncte de lucru într-o zonă geografică. O rețea de șosele leagă direct unele dintre aceste puncte de lucru și între oricare două puncte de lucru există o legătură prin intermediul rețelei de șosele. Distanța dintre două puncte de lucru între care există legătură directă este măsurată în kilometri. Firma trebuie să-și stabilească într-unul din punctele de lucru sediul central. Criteriul de alegere este ca suma distanțelor la celelalte puncte de lucru să fie minimă. Scrieți un program care să stabilească punctul de lucru care va fi ales ca sediu. Datele se citesc dintr-un fișier, astfel: de pe primul rând, numărul de puncte de lucru, iar de pe următoarele rânduri, triplete de numere x, y, d care semnifică faptul că punctele de lucru x și y sunt legate direct de o șosea cu lungimea d . **Indicație.** Pentru fiecare punct de lucru se determină suma distanțelor minime la toate celelalte puncte de lucru, după care se determină suma minimă.
3. O firmă trebuie să colecteze ambalaje din mai multe puncte de lucru din oraș, fiecare punct de lucru găsindu-se pe o anumită stradă. O rețea de intersecții leagă direct unele dintre aceste puncte de lucru și între oricare două puncte de lucru există o legătură prin intermediul traficului pe străzi. Traficul între două intersecții nu este întotdeauna în ambele sensuri, dar între oricare două intersecții există trafic prin intermediul altor intersecții. Cantitatea de ambalaje care poate fi colectată între două intersecții între care există trafic direct este măsurată în kilograme. Scrieți un program care să găsească un traseu optim între două intersecții A și B astfel încât o mașină care pleacă din intersecția A și trebuie să ajungă în intersecția B să colecteze o cantitate cât mai mare de ambalaje. Datele se citesc dintr-un fișier, astfel: de pe primul rând, numărul de intersecții, iar de pe următoarele rânduri, triplete de numere x, y, c care semnifică faptul că intersecția x este legată direct de intersecția y prin traficul de pe o stradă de pe care se poate colecta cantitatea c de ambalaje. Etichetele intersecțiilor A și B se citesc de la tastatură.

4. O firmă se poate aproviziona cu trei sortimente de materiale de la mai multe depozite situate în localități răspândite într-o zonă geografică. Într-un depozit nu este obligatoriu să existe toate cele trei sortimente. O rețea de șosele leagă direct unele dintre aceste localități și între oricare două localități există o legătură prin intermediul rețelei de șosele. Distanța dintre două localități între care există legătură directă este măsurată în kilometri. Firma trebuie să se aprovizioneze cu unul dintre sortimentele de materiale. Scrieți un program care să stabilească depozitul de la care să se aprovizioneze firma astfel încât drumul care trebuie parcurs pentru aprovizionare să aibă lungimea minimă. Fiecare sortiment de material î se atribuie un număr: 1, 2 și 3. Datele se citesc dintr-un fișier text. De pe primul rând se citesc numărul de localități n în care se găsesc depozite și numărul de legături directe între localități, m . De pe următoarele m rânduri se citesc triplete de numere x, y, d care semnifică faptul că localitățile x și y sunt legate direct de o șosea cu lungimea d . De pe fiecare dintre ultimele trei rânduri, în ordinea etichetelor sortimentelor de materiale, se citește câte un sir de numere despărțite prin spațiu cu informații despre locațiile depozitelor unde se găsește acel sortiment de material: primul număr reprezintă numărul de depozite, iar următoarele numere etichetele localităților în care se găsesc aceste depozite. Numărul sortimentului de materiale p se comunică de la tastatură. **Indicație.** Se determină distanța minimă de la localitatea în care se găsește firma la localitatele în care se găsesc depozitele și se alege dintre aceste distanțe cea mai mică distanță care corespunde unui depozit ce conține sortimentul de material p .

2.7.10. Grafuri speciale

Există următoarele grafuri speciale:

- grafuri bipartite;
- grafuri hamiltoniene;
- grafuri euleriene;
- grafuri turneu.

2.7.10.1. Graful bipartit

Graful $G = (X, U)$ se numește **graf bipartit** dacă există două mulțimi nevide de noduri A și B care au următoarele proprietăți: $A \cup B = X$ și $A \cap B = \emptyset$ și orice muchie (arc) din mulțimea U are o extremitate în mulțimea de noduri A și o altă extremitate în mulțimea de noduri B .

Exemple:

1. Graful neorientat G_4 definit anterior este un graf bipartit, deoarece există mulțimile A și B care să îndeplinească condițiile din definiție: $A = \{1, 3, 6, 7, 8, 10\}$ și $B = \{2, 4, 5, 9, 11\}$. Se observă că: $A \cup B = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\} = X_4$ și $A \cap B = \emptyset$ și că fiecare muchie $u \in U_4 = \{[1, 2], [1, 4], [2, 3], [3, 4], [3, 5], [5, 6], [5, 7], [5, 8], [7, 9]\}$ are o extremitate în mulțimea A și celalaltă extremitate în mulțimea B :

$$\begin{array}{lll} [1, 2] \Rightarrow 1 \in A \text{ și } 2 \in B & [1, 4] \Rightarrow 1 \in A \text{ și } 4 \in B & [2, 3] \Rightarrow 2 \in B \text{ și } 3 \in A \\ [3, 4] \Rightarrow 3 \in A \text{ și } 4 \in B & [3, 5] \Rightarrow 3 \in A \text{ și } 5 \in B & [5, 6] \Rightarrow 5 \in B \text{ și } 6 \in A \\ [5, 7] \Rightarrow 5 \in B \text{ și } 7 \in A & [5, 8] \Rightarrow 5 \in B \text{ și } 8 \in A & [7, 9] \Rightarrow 7 \in A \text{ și } 9 \in B \end{array}$$

Observație. Nodurile izolate pot face parte din oricare dintre cele două mulțimi.

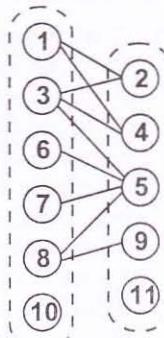


Fig. 35

2. Graful orientat G_9 definit anterior este un graf bipartit deoarece există multimiile A și B care să îndeplinească condițiile din definiție: $A = A = \{1, 3, 4, 6\}$ și $B = \{2, 5\}$. Se observă că: $A \cup B = \{1, 2, 3, 4, 5, 6\} = X_9$ și $A \cap B = \emptyset$ și că fiecare muchie $u \in U_9 = \{[1,2], [2,1], [2,3], [2,4], [3,5], [4,2], [4,5], [5,3], [5,6], [6,5]\}$ are o extremitate în mulțimea A și cealaltă extremitate în mulțimea B:

$$\begin{array}{lll} [1,2] \Rightarrow 1 \in A \text{ și } 2 \in B & [2,1] \Rightarrow 2 \in B \text{ și } 1 \in A & [2,3] \Rightarrow 2 \in B \text{ și } 3 \in A \\ [2,4] \Rightarrow 2 \in B \text{ și } 4 \in A & [3,5] \Rightarrow 3 \in A \text{ și } 5 \in B & [4,2] \Rightarrow 4 \in A \text{ și } 2 \in B \\ [4,5] \Rightarrow 4 \in A \text{ și } 5 \in B & [5,3] \Rightarrow 5 \in B \text{ și } 3 \in A & [5,6] \Rightarrow 5 \in B \text{ și } 6 \in A \\ [6,5] \Rightarrow 6 \in A \text{ și } 5 \in B & & \end{array}$$

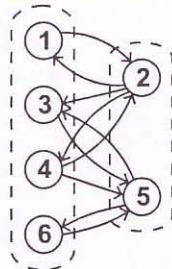


Fig. 36

Graful bipartit $G = (X, U)$ se numește **graf bipartit complet** dacă – pentru orice nod $x_i \in A$ și orice nod $x_j \in B$ – există o muchie (un arc) formată din cele două noduri care aparțin mulțimii U : $[x_i, x_j] \in U$.

Exemple:

1. Graful neorientat $G_{28} = (X_{28}, U_{28})$ definit astfel: $X_{28} = \{1, 2, 3, 4\}$ și $U_{28} = \{[1,2], [1,4], [2,3], [2,3]\}$ este un graf bipartit complet, deoarece există mulțimiile A și B care îndeplinească condițiile din definiție: $A = \{1, 3\}$ și $B = \{2, 4\}$. Se observă că: $A \cup B = \{1, 2, 3, 4\} = X_x$ și $A \cap B = \emptyset$ și că fiecare nod din mulțimea A este legat cu o muchie de fiecare nod din mulțimea B.

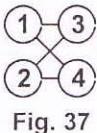


Fig. 37

2. Graful orientat $G_{29} = (X_{29}, U_{29})$ definit astfel: $X_{29} = \{1, 2, 3, 4\}$ și $U_{29} = \{[1,2], [1,4], [2,1], [2,3], [3,2], [4,3]\}$ este un graf bipartit complet, deoarece există mulțimiile A și B care să îndeplinească condițiile din definiție: $A = \{1, 3\}$ și $B = \{2, 4\}$. Se observă că: $A \cup B = \{1, 2, 3, 4\} = X_x$ și $A \cap B = \emptyset$ și că fiecare nod din mulțimea A este legat cu cel puțin un arc de fiecare nod din mulțimea B.

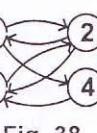


Fig. 38

Observație. Graful neorientat bipartit complet se notează cu $K_{a,b}$ și are $a \times b$ muchii, unde $a = \text{card}(A)$ și $b = \text{card}(B)$. De exemplu, dacă graful neorientat are 3 noduri, se obțin următoarele variante de grafuri bipartite complete:

Elementele mulțimii		Muchiile grafului	Numărul de muchii ale grafului
A	B		
1	2,3	[1,2], [1,3]	$1 \times 2 = 2$
2	1,3	[2,1], [2,3]	$1 \times 2 = 2$
3	1,2	[3,1], [3,2]	$1 \times 2 = 2$
1,2	3	[1,3], [2,3]	$2 \times 1 = 2$
1,3	2	[1,2], [3,2]	$2 \times 1 = 2$
2,3	1	[2,1], [3,1]	$2 \times 1 = 2$
1,2,3	\emptyset	nu există, deoarece $B = \emptyset$	$3 \times 0 = 0$
\emptyset	1,2,3	nu există, deoarece $A = \emptyset$	$0 \times 3 = 0$

Algoritmi pentru prelucrarea grafurilor bipartite

1. Generarea tuturor grafurilor neorientate bipartite complete cu n noduri.

Algoritm. Problema se reduce la a genera toate submulțimile care se pot obține din cele n elemente (exceptând mulțimea inițială și mulțimea vidă). Numărul total de submulțimi obținute este $2^n - 2$. Soluția este de a genera într-un vector nodurile care aparțin mulțimilor A și B, astfel: dacă un element are valoarea 1, nodul care are eticheta corespunzătoare indicelui elementului aparține mulțimii A; altfel, aparține mulțimii B.

Implementarea algoritmului. Funcția `generare()` generează grafurile bipartite complete. În vectorul `a` se generează nodurile mulțimilor A și B. Inițial elementele vectorului au

valoarea 0. Variabila **posibil** se folosește pentru a verifica dacă mai există posibilități de generare de submulțimi (are valoarea 1 – *true*, atunci când mai este posibil să se genereze submulțimi).

```
#include<iostream.h>
#include<math.h>
void generare (int n)
{int a[10]={0},i,j,k=0,posibil=1;
 while (posibil)
 {j=n;
  while (j>0 && a[j]==1) {a[j]=0; j--;}
  if (j==0) posibil=0;
  else {a[j]=1;k++;
    if (k<=pow(2,n)-2)
      {cout<<"Graful "<<k<<endl<<"Multimea A: ";
       for (i=1;i<=n;i++) if (a[i]) cout<<i<<" ";
       cout<<"Multimea B: ";
       for (i=1;i<=n;i++) if (!a[i]) cout<<i<<" ";
       cout<<endl;
       cout<<"Muchiile sunt: ";
       for (i=1;i<=n;i++)
         if (a[i]==1)
           for (j=1;j<=n;j++)
             if (a[j]==0 && i!=j) cout<<"["<<i<<","<<j<<"] ";
       cout<<endl;}}}
void main() {int n; cout<<"numar de noduri= "; cin>>n; generare(n);}
```

2. Verificarea unui graf dacă este graf bipartit.

Algoritmul. Pentru a verifica dacă graful este bipartit, se generează mulțimile de noduri A și B până când aceste mulțimi îndeplinesc condiția unui graf bipartit, sau până când s-au generat toate mulțimile și nici una dintre variante nu a îndeplinit condiția pentru graful bipartit. Graful este bipartit dacă între orice pereche de elemente din cele două mulțimi – (x,y) , cu $x \in A$ și $y \in B$ – există muchie care să le lege în graf ($[x,y] \in U$). Pentru generarea mulțimilor de noduri A și B se pot folosi două variante:

Varianta 1. Ca și în algoritmul precedent, se generează într-un vector toate submulțimile care se pot obține din cele n etichete de noduri (exceptând mulțimea inițială și mulțimea vidă) și se verifică dacă nodurile aparținând celor două mulțimi generate pot fi mulțimile unui graf bipartit.

Implementarea algoritmului. Se citesc din fișierul text *gb.txt* informații despre un graf neorientat (numărul de noduri și matricea de adiacență). Funcția **bipartit()** verifică dacă graful este bipartit furnizând un rezultat logic. Elementele vectorului **x** în care se generează mulțimile A și B sunt inițial 0. Variabila **gasit** se folosește pentru a verifica dacă s-au găsit cele două mulțimi de noduri corespunzătoare unui graf bipartit (are valoarea 1 – *true*, atunci când s-au găsit).

```
#include<fstream.h>
#include<math.h>
fstream f("gb.txt",ios::in);
int a[10][10],n;
void citeste(){//citește matricea de adiacență din fișier}
int bipartit()
{int x[10]={0},i,j,m,k=0,posibil=1,gasit=0;
 while (posibil && !gasit)
```

```

{m=n;
while (m>0 && x[m]==1) {x[m]=0; m--;}
if (m==0) posibil=0;
else
{x[m]=1;k++;
if (k<=pow(2,n)-2)
for (i=1,gasit=1;i<=n && gasit;i++)
for (j=1;j<=n && gasit;j++)
if (a[i][j]==1)
if ((x[i]==1 && x[j]==1) || (x[i]==0 && x[j]==0)) gasit=0;}
return gasit;}
void main() {citeste();
if (bipartit()) cout<<"Este bipartit";
else cout<<"Nu este bipartit";}

```

Varianta 2. Elementele mulțimilor A și B se generează separat, în doi vectori. Pentru generația mulțimii A se folosește **metoda backtracking** (etichetele nodurilor mulțimii A se generează în stivă). Funcția **bt()** este apelată de **n-1** ori pentru a genera în stivă cele p elemente ale mulțimii A ($1 \leq p \leq n-1$). Mulțimea B este formată din nodurile grafului care nu se găsesc în mulțimea A.

Implementarea algoritmului. Matricea A este matricea de adiacență a grafului, iar în vectorii a și b se generează elementele mulțimilor A și B. În funcția **tipar()** se copiază conținutul stivei în vectorul a și se scriu în vectorul b etichetele nodurilor care nu sunt în vectorul a. Pentru a identifica etichetele nodurilor care nu sunt în vectorul a se folosește variabila logică **gasit**. Tot în această funcție se verifică dacă aceste mulțimi corespund unui graf bipartit, astfel: se verifică, pentru toate elementele celor doi vectori, dacă muchiile generate cu un nod din vectorul a și un nod din vectorul b sunt muchii în graf. În caz afirmativ, se verifică dacă muchiile astfel generate sunt toate muchiile grafului. Variabila **nr**, se folosește pentru a număra muchiile formate cu nodurile din cele doi vectori, iar variabila logică **este** pentru a verifica dacă graful este bipartit. (are valoarea 1 – true, dacă numărul de muchii găsite este egal cu numărul total de muchii ale grafului m).

```

#include<iostream.h>
fstream f("qb.txt",ios::in);
typedef int stiva[100];
int n,k,ev,as,A[10][10],b[10],a[10],m,este,p;
stiva st;
void citeste()
{int i,j; f>>n;
for(i=1;i<=n;i++)
for (j=1;j<=n;j++) {f>>A[i][j]; m=m+A[i][j];} f.close(); m=m/2;}
void init(){st[k]=0;}
int succesor ()
{if (st[k]<n) {st[k]=st[k]+1; return 1;} else return 0;}
int valid()
{if (k>1 && st[k]<st[k-1]) return 0;
for (int i=1;i<k;i++) if (st[k]==st[i]) return 0;
return 1;}
int solutie() {return k==p;}
void tipar()
{int i,j,q=0,gasit,nr=0;
for (i=1;i<=p;i++) a[i]=st[i];
for (i=1; i<=n; i++)

```

```

    {for (j=1,gasit=0; j<=p && !gasit; j++)
        if (!gasit) {q++; b[q]=i;}
    for (i=1; i<=p; i++)
        for (j=1; j<=q; j++) if (A[a[i]][b[j]]==1) nr++;
    if (nr==m) este=1;}
void bt() {//partea fixa a algoritmului backtracking}
void main() {citeste();
            for (p=1; p<=n-1 && !este; p++) bt();
            if (este) cout<<"Este bipartit";
            else cout<<"Nu este bipartit";}
    
```

Temă

1. Scrieți un program care citește din fișierul *graf_b1.txt* matricea de adiacență a unui graf neorientat și, de la tastatură, două siruri de numere ce reprezintă noduri din graf – și care verifică dacă cele două siruri de numere pot reprezenta pentru graf cele două mulțimi ale unui graf bipartit.
2. Comparați din punct de vedere al eficienței cei doi algoritmi prin care se determină dacă un graf neorientat este bipartit.
3. Scrieți un program care citește din fișierul *graf_b2.txt* matricea de adiacență a unui graf orientat și care verifică dacă graful este bipartit. În caz afirmativ, se afișează mulțimile de noduri A și B.

2.7.10.2. Graful hamiltonian

Într-un graf $G=(X,U)$, se numește **lanț hamiltonian** lanțul elementar care conține toate nodurile grafului.

Altfel spus, un lanț este hamiltonian dacă pornește de la un nod oarecare și parcurge o singură dată toate nodurile grafului.

Lanțul hamiltonian în care nodul inițial coincide cu nodul final se numește ciclu hamiltonian.

Ca într-un graf să existe un ciclu hamiltonian este necesar ca pe lângă un lanț hamiltonian să mai existe și o mulțime care să lege primul nod al lanțului de ultimul nod al lanțului.

Un graf care conține un ciclu hamiltonian se numește graf hamiltonian.

Altfel spus, un graf hamiltonian este un graf în care – pornind de la un nod oarecare – se pot parcurge o singură dată toate nodurile grafului, revenind la nodul inițial.

G₃₀

Graful G_{30} din figura 39 conține ciclul hamiltonian $C = \{1, 2, 3, 6, 4, 5, 1\}$.

Observație. Un graf hamiltonian nu poate conține noduri izolate.

Propoziția 10

Graful complet K_n este hamiltonian.

Demonstrație. Graful complet K_n poate fi privit ca un poligon cu n laturi, în care fiecare vârf al poligonului este legat de celelalte vârfuri prin diagonale, iar vârfurile poligonului parcurse pe laturi formează un ciclu hamiltonian.

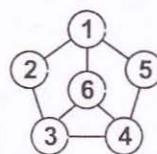


Fig. 39

Studiu de caz

Scop: exemplificarea unei aplicații în care soluția problemei este un graf hamiltonian.

Problema voiajorului comercial. Un voiajor comercial trebuie să călăorească în mai multe orașe pentru a-și prezenta produsele. Se cunoaște costul deplasării între localități. Trebuie să se stabilească un traseu prin care să se ajungă în toate localitățile, astfel încât

costul deplasării să fie minim. Voiatorul va trece o singură dată prin fiecare localitate și se va întoarce în localitatea de pornire.

Localitățile formează nodurile unui graf, iar traseul voiajorului trebuie să fie un ciclu hamiltonian în acest graf. Dacă există mai multe cicluri hamiltoniene, se va alege cel cu costul minim (ciclul în care suma costurilor asociate muchiilor este minimă).

Observație. Orice problemă la care soluția este de a găsi un traseu – care pornește dintr-un anumit punct, trebuie să treacă prin puncte precizate, cu revenire la punctul de pornire – se rezolvă prin găsirea unui ciclu hamiltonian. De exemplu:

- O firmă de mesagerie trebuie să distribue zilnic colete la mai multe adrese. Mașina care distribue aceste colete pleacă de la sediul firmei, ajunge la mai multe puncte din oraș și revine la sediul firmei. Legătura directă dintre două puncte este caracterizată prin distanță măsurată în kilometri (costul asociat fiecărei muchii). Trebuie să se găsească traseul de lungime minimă pe care trebuie să-l parcurgă mașina.
- O persoană dorește să facă un circuit prin țară și să viziteze mai multe puncte turistice, plecând din localitatea de domiciliu și întorcându-se în aceeași localitate. Legătura directă dintre două puncte turistice este caracterizată prin distanță măsurată în kilometri (costul asociat fiecărei muchii). Trebuie să se găsească circuitul turistic de lungime minimă.
- O persoană dorește să viziteze mai multe cabane, întorcându-se la locul de plecare. Legătura directă dintre două cabane este caracterizată prin timpul necesar parcurgerii traseului de munte (costul asociat fiecărei muchii). Trebuie să se găsească circuitul de vizitare a cabanelor care să se facă în timp minim.



Teorema 18

Dacă graful $G=(X,U)$ este un graf cu mai mult de două noduri ($n \geq 3$) și gradul fiecărui nod $x \in X$ satisfacă condiția $d(x) \geq n/2$, atunci graful G este hamiltonian.

Observație. Această teoremă precizează numai **condiția suficientă** ca un graf să fie hamiltonian. Această condiție nu este însă și necesară. Astfel, este posibil ca un graf să nu îndeplinească această condiție și să fie totuși hamiltonian. Din această cauză, teorema nu poate fi folosită pentru a construi un algoritm care să verifice dacă un graf este hamiltonian.

Demonstrație. Notăm cele două propoziții, astfel:

- (1) – Graful G are mai mult de două noduri și gradul fiecărui nod $x \in X$ satisfacă condiția $d(x) \geq n/2$.
- (2) – Graful G este hamiltonian.

Trebuie să demonstrăm că (1) \Rightarrow (2) – prin reducere la absurd. Presupunem că graful G nu este hamiltonian. Adăugăm muchii la acest graf până se obține un graf hamiltonian (în cel mai rău caz se ajunge la un graf complet, care sigur este hamiltonian). Prin adăugarea de noi muchii, gradul unor noduri crește, ceea ce înseamnă că se păstrează inegalitatea din propoziția (1). Adăugarea acestor muchii se face până când se obține graful G_1 care nu este hamiltonian, dar care, prin adăugarea unei singure muchii devine hamiltonian. Acest graf nu este sigur un graf complet și înseamnă că există cel puțin două noduri neadiacente. Luăm dintre aceste noduri două noduri x și y și $[x,y] \notin U_1$ care au proprietatea că, prin adăugarea muchiei $[x,y]$ la graful G_1 , se obține un ciclu hamiltonian $C = \{x, x_1, \dots, x_{k-1}, x_k, y, x\}$. Însămnă că în graful G_1 există un lanț elementar $L(x,y)$ care conține toate nodurile grafului. Deoarece $d(x) \geq n/2$, rezultă că în graful G_1 mai există un nod x_i cu care este adiacent. Acest nod face parte din lanțul $L(x,y)$. Fie x_{i-1} nodul care precede nodul x_i în lanțul $L(x,y)$. Dacă nodurile x_{i-1} și y formează o muchie $[x_{i-1},y] \in U_1$, înseamnă că graful G_1 conține un ciclu hamiltonian $C_1 = \{x, x_1, \dots, x_{i-1}, y, x_k, x_{k-1}, \dots, x_i, x\}$, ceea ce contrazice presupunerea că graful G_1 nu este hamiltonian. Rezultă că nodul y nu formează muchie cu nici unul dintre nodurile care îl precedă în lanțul $L(x,y)$ și care sunt adiacente cu nodul x_i . Înseamnă că nodul y nu poate fi adiacent decât cu cel mult $n-1-d(x_i)$ noduri, adică $d(y) \leq n-1-d(x_i)$. Dar, $d(x_i) \geq n/2$. Rezultă că $d(y) \leq n-1-n/2=n/2-1$, adică $d(y) \leq n/2$, ceea ce contrazice ipoteza. Contradicția a apărut din presupunerea că graful G nu este hamiltonian.

Temă

1. Verificați dacă graful hamiltonian G_{30} din figura 39 îndeplinește condițiile precizate în teoremă.
 2. Desenați toate grafurile hamiltoniene cu 4 noduri.
- G₃₁** 3. Verificați dacă este hamiltonian graful $G_{31}=(X_{31}, U_{31})$ cu $X_{31}=\{1,2,3,4,5,6,7\}$ și $U_{31}=\{[1,2], [1,5], [1,6], [1,7], [2,3], [2,7], [3,4], [3,5], [3,7], [4,5], [4,6], [5,6]\}$. Dacă este graf hamiltonian, găsiți un ciclu hamiltonian și verificați dacă graful îndeplinește condițiile precizate în teoremă.
4. Demonstrați că un graf bipartit cu număr impar de noduri nu poate fi graf hamiltonian.
 5. Demonstrați că numărul ciclurilor hamiltoniene din graful complet cu n noduri este $\frac{(n-1)!}{2}$.

Algoritmul pentru parcurgerea unui graf hamiltonian

Algoritmul. Pentru a determina dacă un graf este hamiltonian se verifică dacă există un ciclu hamiltonian. Este suficient să se caute lanțurile elementare care pornesc din nodul cu eticheta 1 și se închid în acest nod. Se poate folosi fie metoda **backtracking**, fie metoda **parcurgerii în lățime a grafului**. Prin aceste metode se pot determina și toate ciclurile hamiltoniene, dacă există, astfel: se caută toate ciclurile elementare care, pornind dintr-un nod, parcurează toate celelalte noduri ale grafului și se închid printr-o muchie cu nodul de pornire.

Implementarea algoritmului. Se citește din fișierul *graf_h.txt* matricea de adiacență a unui graf neorientat. Dacă graful este hamiltonian, se afișează ciclurile hamiltoniene găsite. Dacă nu există nici o soluție, se afișează un mesaj de informare. Pentru generarea lanțurilor elementare se folosește metoda **backtracking**. Nodurile lanțului elementar vor fi generate în stivă. Funcția **citere()** se folosește pentru a citi matricea de adiacență în fișier. Variabila **este** se folosește pentru a verifica dacă s-a găsit un ciclu hamiltonian pornind din nodul cu eticheta 1 (are valoarea 0 – *False*, dacă nu s-a găsit un ciclu hamiltonian).

```
#include<iostream.h>
typedef stiva[100];
int a[20][20],n,k,as,ev,este=0;
stiva st;
fstream f("graf_h.txt",ios::in);
void citeste() { //se citește matricea de adiacență}
void init() {st[k]=0;}
int succesor()
{if (st[k]<n) {st[k]=st[k]+1; return 1;} else return 0;}
int valid()
{if(k>1 && a[st[k-1]][st[k]]==0) return 0;
 for (int i=1;i<k;i++) if (st[k]==st[i]) return 0;
return 1;}
int solutie() {return a[st[k]][1]==1 && k==n;}
void tipar()
{for (int i=1,este=1;i<=n;i++) cout<<st[i]<<", "; cout<<st[1]<<\endl;}
void bt() { //identică cu cea de la generarea lanțurilor elementare}
void main() {citeste(); st[1]=1; bt();
 if (!este) cout<<"Graful nu este hamiltonian";}
```

Temă

1. Scrieți un program care citește din fișierul text *graf_h1.txt* matricea de adiacență a unui graf hamiltonian, verifică dacă graful îndeplinește condiția precizată în teoremă pentru un graf hamiltonian și verifică dacă este hamiltonian – folosind metoda parcurgerii în lățime a grafului.
2. Scrieți un program care citește din fișierul text *graf_h2.txt* matricea de adiacență a unui graf hamiltonian și, de pe ultimul rând, un sir de $n+1$ numere care reprezintă etichete de noduri și care să verifică dacă sirul de etichete reprezintă un ciclu hamiltonian.

2.7.10.3. Graful eulerian

Într-un graf $G=(X,U)$, se numește **lanț eulerian** lanțul care conține toate muchiile grafului, fiecare muchie fiind prezentă o singură dată.

Un lanț este eulerian dacă pornește de la un nod oarecare și parcurge o singură dată toate muchiile grafului (este un **lanț simplu** care parcurge toate muchiile grafului).

Lanțul eulerian în care nodul inițial coincide cu nodul final se numește ciclu eulerian.

Ca într-un graf să existe un ciclu eulerian este necesar ca, pe lângă un lanț eulerian, să mai existe și o muchie care să lege primul nod al lanțului de ultimul nod al lanțului, și acea muchie să nu mai fi fost parcursă.

Un graf care conține un ciclu eulerian se numește graf eulerian.

Altfel spus, un graf eulerian este un graf în care, pornind de la un nod oarecare se pot parcurge o singură dată toate muchiile grafului, revenind la nodul inițial.

Graful G_{32} din figura 40 conține ciclul eulerian $C = \{1, 4, 6, 7, 4, 5, 7, 8, 3, 2, 8, 5, 2, 1\}$.

G_{32}

Observații

1. Un graf eulerian **poate contine noduri izolate**.
2. Pentru ca un graf să poată fi făcut eulerian prin adăugarea muchiilor trebuie să fie îndeplinite următoarele condiții:

→ dacă numărul de noduri este par, să nu existe noduri cu gradul maxim;
→ numărul de noduri cu grad impar să fie par.

Numărul minim de muchii care trebuie adăugate este egal cu jumătate din numărul de noduri cu grad impar.

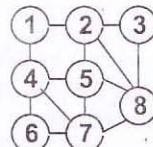


Fig. 40

Studiu de caz

Scop: exemplificarea aplicației în care soluția problemei este un graf eulerian.

Problema podurilor din orașul Königsberg.

În acest oraș există șapte poduri peste delta râului Pregel care leagă cele patru sectoare ale orașului (figura 41). Problema constă în a stabili un traseu prin care un vizitator să parcurgă toate cele patru sectoare ale orașului (A, B, C și D), întorcându-se în punctul de unde a plecat, trecând peste fiecare pod o singură dată.



Fig. 41

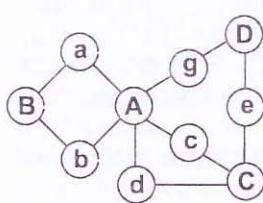


Fig. 42

Cele patru zone (notate cu A, B, C și D) și cele șapte poduri (notate cu a, b, c, d, e, și f) reprezintă nodurile unui graf neorientat G_{33} (figura 42), iar muchiile reprezintă posibilitatea de a trece pe un pod dintr-o zonă, în altă zonă. Problema constă în a găsi în acest graf un ciclu care să parcurgă toate muchiile o singură dată, adică un ciclu eulerian. Matematicianul Euler a demonstrat matematic, în anul 1736, că această problemă nu are soluții (că acest graf nu este eulerian).



G_{33}

Teorema 19

Un graf $G=(X,U)$, care nu conține noduri izolate, este eulerian dacă și numai dacă este conex și gradele tuturor nodurilor sunt numere pare.

Demonstrație. Notăm cele două propoziții, astfel:

- (1) – Graful G nu conține noduri izolate și este eulerian.
- (2) – Graful G este conex și gradele tuturor nodurilor sunt numere pare.

Trebuie să demonstrăm că $(1)\Rightarrow(2)$ și $(2)\Rightarrow(1)$.

$(1)\Rightarrow(2)$

a. **Graful este conex.** Trebuie să demonstrăm că, oricare ar fi nodurile x și y din graf, există un lanț $L(x,y)$ care le leagă. Pentru cele două noduri poate să apară una dintre următoarele situații:

- Sunt adiacente $([x,y]\in U)$. Înseamnă că există lanțul $L(x,y)$.
- Nu sunt adiacente $([x,y]\notin U)$. Deoarece graful nu are noduri izolate, înseamnă că există alte două noduri z și w din graf care sunt adiacente cu câte unul dintre ele $([x,z]\in U$ și $[y,w]\in U)$. Deoarece graful este eulerian, există un ciclu care conține toate muchiile grafului, inclusiv muchiile $[x,z]$ și $[y,w]$. Fie acest ciclu $C = \{x_1, \dots, x, \dots, y, \dots, x_k\}$. Acest ciclu conține un lanț $L(x,y)$.

Deoarece în ambele situații există un lanț $L(x,y)$, înseamnă că graful este conex.

b. **Gradele tuturor nodurilor sunt numere pare.** Trebuie să demonstrăm că pentru un nod oarecare $x\in X$, $d(x)$ este un număr par. Ciclul eulerian conținând toate muchiile grafului, conține și toate nodurile grafului. Acest ciclu se poate reorganiza astfel încât nodul x să nu fie primul nod: $C = \{y, \dots, x, \dots, y\}$. Nodul x poate să apară în acest ciclu de p ori. Fiecare apariție a nodului x în acest ciclu înseamnă existența a două muchii diferite incidente cu el. Rezultă că $d(x)=2p$, adică un număr par.

$(2)\Rightarrow(1)$ – prin reducere la absurd.

Presupunem că graful G nu este eulerian. Graful fiind conex, nu conține noduri izolate. Fie două noduri oarecare x și y între care există o muchie $([x,y]\in U)$. Nodul y având gradul un număr par, înseamnă că există un nod z cu care formează o muchie $([y,z]\in U)$. Continuând în acest mod, se ajunge la un nod w care formează muchie cu nodul x $([w,x]\in U)$, deoarece și gradul nodului x este un număr par. Rezultă că în graful G există un ciclu. Vom considera, dintre ciclurile care există în acest graf, ciclu C de lungime maximă, dar care, conform presupunerii făcute, nu conține toate muchiile grafului. Pentru fiecare nod din acest ciclu se consumă două unități din gradul lui. Căutăm o muchie care nu face parte din ciclul C , dar este incidentă cu un nod din ciclul C . Această muchie există deoarece dacă nu ar fi, ciclul C ar forma o componentă conexă, ceea ce contrazice ipoteza că graful G este conex. Fie muchia $[x,y]\in U$, cu $x\in C$. Pornind de la această muchie, pe muchii care nu aparțin ciclului C , trecând prin noduri ale căror grade nu au fost epuizate, se obține, după un număr finit de pași, un nou ciclu C_1 care se închide în nodul x . Prin concatenarea celor două cicluri care au un nod comun (x), se obține un ciclu de lungime mai mare decât ciclul C , ceea ce contrazice presupunerea că ciclul C are cea mai mare lungime. Înseamnă că el conține toate muchiile grafului și graful este eulerian.

Observație. Această teoremă, precizând **condițiile necesare și suficiente** ca un graf să fie eulerian, poate fi folosită pentru a construi un algoritm care să verifice dacă un graf este eulerian.

G₃₄

Tema



1. Verificați condițiile din teoremă pentru graful eulerian G_{32} din figura 40.
2. Verificați dacă graful $G_{34}=(X_{34},U_{34})$ cu $X_{34}=\{1,2,3,4,5,6,7\}$ și $U_{34}=\{[1,2], [1,3], [2,3], [2,4], [2,5], [3,5], [3,7], [4,5], [4,6], [4,7], [5,6]\}$ este eulerian. Dacă graful este eulerian, găsiți un ciclu eulerian și verificați dacă graful îndeplinește condițiile precizate în teoremă.
3. Demonstrați că într-un graf conex G există două noduri diferențiate x și y legate printr-un lanț eulerian – dacă și numai dacă ele sunt singurele noduri cu grad impar din graf.
4. Desenați toate grafurile euleriene cu 4 noduri.
5. Verificați dacă graful conex $G_{35}=(X_{35},U_{35})$ cu $X_{35}=\{1,2,3,4,5,6,7,8\}$ și $U_{35}=\{[1,2], [1,4], [2,3], [2,4], [2,5], [3,6], [4,5], [4,7], [5,6], [5,7], [5,8], [6,8], [7,8]\}$ poate fi făcut eulerian prin adăugare de muchii. În caz afirmativ, precizați căte muchii trebuie adăugate și care sunt aceste muchii.

G₃₅

6. Verificați dacă graful neconex $G_{36} = (X_{36}, U_{36})$ cu $X_{36} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$ și $U_{36} = \{[1, 2], [2, 4], [3, 4], [4, 5], [4, 6], [5, 6], [5, 7], [6, 7], [8, 9], [8, 11], [9, 10], [9, 11], [10, 11]\}$ poate fi făcut eulerian prin adăugare de muchii. În caz afirmativ, precizați câte muchii trebuie adăugate și care sunt aceste muchii. Găsiți un algoritm pentru generalizarea problemei.
7. Să se dea exemple de următoarele grafuri:
- să nu fie hamiltonian și nici eulerian;
 - să fie hamiltonian, dar nu eulerian;
 - să nu fie hamiltonian, dar să fie eulerian;
 - să fie hamiltonian și eulerian.

Algoritmul pentru parcurgerea unui graf eulerian

Pentru a implementa graful se folosește matricea de adiacență a și vectorul g în care se memorează gradul fiecărui nod, care se calculează cu funcția `grad()`. **Algoritmul care determină dacă un graf este eulerian** verifică dacă graful îndeplinește condițiile precizate în teoremă:

- **Nodurile izolate.** Se verifică dacă graful are sau nu noduri izolate. Dacă are noduri izolate se consideră că graful nu este eulerian. În implementarea algoritmului se folosește funcția `izolat()` care testează gradul nodurilor și furnizează un rezultat logic: `true` (1) dacă cel puțin un nod are gradul 0 și `false` (0) dacă nici un nod nu are gradul 0.
- **Gradul nodurilor.** Se verifică dacă gradul fiecărui nod este par. În implementarea algoritmului se folosește funcția `grad_par()` care furnizează un rezultat logic: `false` (0) dacă cel puțin un nod are gradul impar și `true` (1) dacă nici un nod nu are gradul impar.
- **Conexitatea.** Se verifică dacă graful este conex. Pentru aceasta se va parcurge graful în adâncime și se verifică dacă rămân noduri care nu au fost vizitate. În implementarea algoritmului se folosește vectorul `vizitat` pentru a ține evidența nodurilor vizitate. Conexitatea se verifică prin funcția `conex()` care furnizează un rezultat logic: `false` (0) dacă cel puțin un nod nu a fost vizitat și `true` (1) dacă toate nodurile au fost vizitate.

Pentru testarea condițiilor necesare ca un graf să fie eulerian se folosește variabila `eulerian` care are valoarea logică: `false` (0) dacă graful nu este eulerian și `true` (1) dacă graful este eulerian.

Dacă graful este eulerian se poate **determina un ciclu eulerian**. Acest algoritm folosește demonstrația de la teorema precedentă. Construirea ciclului eulerian se face astfel:

- PAS1.** Se pornește dintr-un nod oarecare din graf și se construiește din aproape în aproape ciclul **C**, pe muchii incidente care există în graf, scăzând gradele nodurilor prin care trece și eliminând muchiile.
- PAS2.** Cât timp mai există muchii care nu fac parte din ciclul **C** execută: se alege un nod din ciclul **C** pentru care mai există muchii incidente care nu fac parte din ciclul **C**, se construiește ciclul **C1** și se concatenează ciclul **C1** la ciclul **C**.

Soluția se va obține într-un vector `c` care conține nodurile parcuse care au fost adăugate la ciclul eulerian. Vectorul `c1` se folosește pentru extinderea ciclului eulerian. Se mai folosesc următoarele variabile de memorie:

- n – numărul de noduri și m – numărul de muchii;
- i, j, k – contori pentru parcurgerea matricei de adiacență și a vectorilor folosiți;
- q – variabilă în care se păstrează nodul de la care începe ciclul `c1`;
- p – variabilă în care se păstrează lungimea logică a vectorului `c1`.

Algoritmul pentru determinarea ciclului eulerian este următorul:

- PAS1.** Se citesc valorile pentru variabilele de memorie n și m și matricea de adiacență a .
- PAS2.** Se verifică dacă graful este eulerian, adică dacă îndeplinește condiția să nu aibă noduri izolate, nodurile să aibă grade pare și să fie conexe: (în implementarea algoritmului: `eulerian!=!v_izolat()&&grad_par()&& conex()`).
- PAS3.** Dacă graful nu este eulerian (variabila `eulerian` are valoarea `false`), atunci se afișează mesajul 'Graful nu este eulerian' și se termină algoritmul; altfel, se scrie mesajul 'Graful este eulerian' și se trece la Pasul 4 pentru a determina un ciclu eulerian.
- PAS4.** Se pleacă dintr-un nod oarecare (de exemplu, nodul 1: `c[1]=1`).
- PAS5.** Execută următorii pași pentru a construi în vectorul `c` un prim ciclu prin parcugerea din aproape în aproape a muchiilor grafului, pornind de la nodul cu eticheta j egală cu 1: $j=1$ (acest ciclu există, deoarece gradul nodului de pornire este par, ceea ce înseamnă că oricare ar fi muchia pe care se pornește, mai există cel puțin o muchie care sosește în acest nod).
- PAS6.** Cât timp nu s-a găsit o muchie între nodul curent k din coada `c` (`c[k]`) și un alt nod j din graf ($j \leq n$) execută:
- PAS7.** Dacă există o muchie între nodul curent k din coada `c` (`c[k]`) și un alt nod j din graf ($a[c[k]][j]==1$), atunci se trece la Pasul 8; altfel, se trece la Pasul 11.
- PAS8.** Se adaugă nodul j la coada `c` (`k=k+1; c[k]=j;`).
- PAS9.** Se micșorează cu o unitate gradul celor două noduri parcuse (`g[j]--; g[c[k-1]]--;`).
- PAS10.** Se șterge din matricea de adiacență muchia parcursă (`a[c[k]][j]=0; a[j][c[k]]=0;`).
- PAS11.** Se trece la următorul nod prin incrementarea lui j ($j=j+1$) și se revine la Pasul 6.
- Până când nodul curent din coada `c` (`c[k]`) este diferit de nodul de pornire.
- PAS12.** Cât timp mai există muchii care nu au fost incluse în ciclu ($k-1 < m$) execută
- PAS13.** Se caută în coada `c` un nod i de plecare pentru ciclul `c1`, adică un nod pentru care mai există muchii incidente cu el care nu au fost parcuse (`grad[c[i]]>0`).
- PAS14.** Se inițializează cu acest nod ciclul `c1` (`c1[1]= c[i]`) și se memorează acest nod în variabila `q`.
- PAS15.** Se construiește un nou ciclu parcurgând – din algoritm – pentru coada `c1`, de la Pasul 5 până la Pasul 11. Acest ciclu va avea p elemente.
- PAS16.** Se concatenează ciclul `c1` cu ciclul `c` astfel: mai întâi se deplasează elementele din vectorul `c`, începând din poziția `q`, cu p poziții spre dreapta, după care se intercalează, între poziția `q` și poziția `q+p`, elementele vectorului `c1`.
- PAS17.** Se afișează elementele vectorului `c`.

```
#include<iostream.h>
typedef stiva[20];
int n,a[20][20],vizitat[20],vf,k,m,g[20],p=1,c[20],c1[20];
stiva st;
fstream f("graf_e.txt",ios::in);
void citeste() { //se citește matricea de adiacență}
void init(int i) {vf=1; st[vf]=i; vizitat[i]=1;}
int este_vida() {return vf==0;}
void adaug(int i) {vf++; st[vf]=i; vizitat[i]=1;}
void elimin() {vf--;}
```

```

void prelucrare()
{int i=1; k=st[vf];
 while (i<=n && (a[i][k]==0 || (a[i][k]==1 && vizitat[i]==1))) i++;
 if (i==n+1) elimin(); else {p++; adaug(i);}}
int conex()
{k=1; init(k);
 while (!este vida()) prelucrare();
 return (p==n);}
void grad()
{for(int i=1;i<=n;i++)
 for (int j=1;j<=n;j++) if (a[i][j]==1) {g[i]++;m++;} m=m/2;}
int izolat()
{for(int i=1;i<=n;i++) if (g[i]==0) return 1;
 return 0;}
int grad_par()
{for(int i=1;i<=n;i++) if (g[i]%2==1) return 0;
 return 1;}
void ciclu()
{int i,j,k=1,p,q,gasit;
 c[1]=1;
 do for(j=1,gasit=0;j<=n && !gasit;j++)
 if (a[c[k]][j]==1)
 {k=k+1; c[k]=j; a[c[k-1]][j]=0; a[j][c[k-1]]=0;
 g[j]--; g[c[k-1]]--; gasit=1;}
 while(c[k]!=1);
 while (k-1<m)
 {for(i=1,q=0;i<=k-1 && q==0;i++)
 if (g[c[i]]>0) {c1[1]=c[i]; q=i;}
 p=1;
 do for(j=1,gasit=0;j<=n && !gasit;j++)
 if (a[c1[p]][j]==1)
 {p=p+1; c1[p]=j; a[c1[p-1]][j]=0; a[j][c1[p-1]]=0;
 g[j]--; g[c1[p-1]]--; gasit=1;}
 while(c1[p]!=c1[1]);
 for(j=k;j>=q;j--) c[j+p-1]=c[j];
 for(j=1;j<=p-1;j++) c[j+q]=c1[j+1];
 k=k+p-1;}}
void main()
{int eulerian; citeste(); grad();
 eulerian!=!(izolat()) && grad_par() && conex();
 if (!eulerian) cout<<"graful nu este eulerian ";
 else {cout<<"graful este eulerian"<<endl;
 ciclu(); cout<<"cicul eulerian este: ";
 for(int i=1;i<=m+1;i++) cout<<c[i]<<" ";}
}

```

Observație. Dacă graful conține noduri izolate, algoritmii anteriori se pot aplica pe subgraful conex care se obține după înlăturarea nodurilor izolate.



1. Determinați complexitatea algoritmului de găsire a unui ciclu eulerian.
2. Scrieți un program care citește din fișierul text *graf_e1.txt* matricea de adiacență a unui graf eulerian și, de pe ultimul rând, un sir de numere care reprezintă etichete de noduri – și care verifică dacă sirul de etichete reprezintă un ciclu eulerian.

3. Scrieți un program care citește din fișierul text *graf_e2.txt* matricea de adiacență a unui graf neorientat care poate conține noduri izolate și care verifică dacă graful este un graf eulerian, iar dacă este graf eulerian, afișează ciclul eulerian.
4. Scrieți un program care citește din fișierul text *graf_e3.txt* matricea de adiacență unui graf neorientat conex și care verifică dacă graful este un graf eulerian, iar dacă nu este, verifică dacă poate fi făcut eulerian – în acest caz, se precizează numărul de muchii care trebuie adăugate și care sunt aceste muchii.

2.7.10.4. Graful turneu

Un graf orientat în care, între oricare două noduri există un singur arc și numai unul, se numește **graf turneu**.

G₃₇

Exemplu – Graful G₃₇ din figura 43

Observații

1. Arcul dintre două noduri poate avea oricare dintre cele două orientări.
2. Graful turneu este un **graf complet**.

Teorema 20

Orice **graf turneu** conține un **drum elementar** care trece prin **toate nodurile** grafului.

Demonstrație – prin reducere la absurd. Presupunem că nu există un drum elementar care trece prin toate nodurile grafului. Considerăm D(x_i, x_j) un drum de lungime maximă din graf și un nod x \notin D(x_i, x_j). Graful turneu fiind un graf complet, înseamnă că nodul x este adiacent cu orice nod din drumul D(x_i, x_j), inclusiv cu nodurile x_i și x_j. Pot să apară următoarele situații:

- Există arcul [x, x_i]. În acest caz, înseamnă că există drumul D(x, x_i) mai lung decât drumul D(x_i, x_j), ceea ce contrazice ipoteza.
- Există arcul [x_j, x]. În acest caz, înseamnă că există drumul D(x_j, x) mai lung decât drumul D(x_i, x_j), ceea ce contrazice ipoteza.
- Nu există arcele [x, x_i] și [x_j, x]. Deoarece graful este complet, înseamnă că există arcele [x_i, x] și [x_j, x]. Dacă pentru orice nod x_k \in D(x_i, x_j) există arce numai cu sensul [x_k, x], înseamnă că există drumul D₁(x_i, x_j) mai lung decât drumul D(x_i, x_j), deoarece conține în plus muchiile [x_{j-1}, x] și [x, x_j]. Dacă pentru orice nod x_k \in D(x_i, x_j) există arce numai cu sensul [x, x_k], înseamnă că există drumul D₂(x_i, x_j) mai lung decât drumul D(x_i, x_j), deoarece conține în plus muchiile [x_i, x] și [x, x_{i+1}]. Dacă există două noduri adiacente x_k și x_{k+1}, care aparțin drumului D(x_i, x_j) pentru care arcele incidente cu nodul x au sensuri contrare, se obține un drum D₃(x_i, x_j) mai lung decât drumul D(x_i, x_j), deoarece conține și arcele [x_k, x] și [x, x_{k+1}], ceea ce contrazice ipoteza.

Rezultă că drumul D(x_i, x_j) conține toate nodurile din graf.

Propoziția 10

Pentru orice graf turneu, **există un nod x**, astfel încât **toate nodurile y ≠ x sunt accesibile din x pe un drum care conține un arc sau două arce**.

Demonstrație – prin reducere la absurd. Considerăm nodul x cu gradul extern maxim – p. Înseamnă că între nodul x și celelalte p noduri există un drum format dintr-un singur arc. Să presupunem că printre celelalte noduri există un nod y care nu este accesibil din nodul x prin două arce. Înseamnă că din acest nod pleacă p arce către nodurile care sunt accesibile printr-un singur arc din nodul x și un arc către nodul x. Înseamnă că nodul y are gradul p+1, mai mare decât gradul nodului y, ceea ce contrazice ipoteza.

Propoziția 11

Pentru orice nod x dintr-un graf turneu cu n noduri, $d^+(x) + d^-(x) = n-1$.

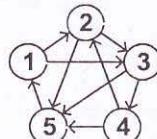


Fig. 43

Demonstrație. În graful turneu, fiecare nod x_i fiind legat de celelalte $n-1$ noduri x_j din graf printr-un arc și numai unul, înseamnă că orice nod x_i este incident cu $n-1$ arce și suma dintre gradul intern și gradul extern este egală cu $n-1$.

Propoziția 12

Într-un graf turneu cu n noduri $\sum_{i=1}^n d^+(x_i) = \sum_{i=1}^n d^-(x_i) = C_n^2$, unde x_i este un nod din graf.

Demonstrație. În orice graf neorientat suma dintre gradele interne și gradele externe este egală cu numărul de arce m ; iar numărul de arce într-un graf turneu se determină la fel ca și în cazul unui graf complet neorientat K_n .

Propoziția 13

Într-un graf turneu $\sum_{i=1}^n d^+(x_i)^2 = \sum_{i=1}^n d^-(x_i)^2$, unde x_i este un nod din graf.

Demonstrație. $d^+(x_i)^2 + d^-(x_i)d^+(x_i) = (n-1)d^+(x_i)$ și $d^-(x_i)^2 + d^+(x_i)d^-(x_i) = (n-1)d^-(x_i)$. Scăzând cele două identități, obținem $d^+(x_i)^2 - d^-(x_i)^2 = (n-1)(d^+(x_i) - d^-(x_i))$. Adunând această identitate pentru toate nodurile x_i obținem:

$$\sum_{i=1}^n d^+(x_i)^2 - \sum_{i=1}^n d^-(x_i)^2 = (n-1) \times (\sum_{i=1}^n d^+(x_i) - \sum_{i=1}^n d^-(x_i)) = (n-1) \times (m - m) = 0$$

Temă

1. Demonstrați că un graf turneu este tare conex, dacă și numai dacă conține un ciclu elementar care trece prin toate nodurile grafului.
2. Scrieți un program care citește din fișierul *text graf_t.txt* matricea de adiacență a unui graf orientat și care verifică dacă graful este un graf turneu.
3. Demonstrați că numărul de grafuri turneu care se pot construi cu n noduri este $2^{C_n^2}$.

Studiu de caz

Scop: exemplificarea unei aplicații în care soluția problemei este un graf turneu.

Enunțul problemei. Pentru un concurs hipic s-au montat n obstacole. Pentru a parcurge traseul concursului, călăreții pot începe cu orice obstacol, trebuie să treacă peste toate obstacolele, iar de la un obstacol la altul pot circula într-un singur sens, stabilit înaintea concursului. Să se precizeze ce trasee poate urma un călăreț.

Obstacolele formează nodurile unui graf orientat, în care fiecare două noduri sunt legate de un arc. Deoarece drumul dintre două obstacole nu poate fi parcurs decât într-un singur sens, înseamnă că graful concursului este un graf turneu. A determina un traseu pentru parcurgerea obstacolelor, înseamnă a determina în graful turneu un drum elementar care trece prin toate nodurile grafului.



Algoritmul pentru parcurgerea grafului turneu

Pentru a determina drumul elementar care trece prin toate nodurile grafului se poate folosi fie metoda backtracking, fie următorul algoritm, prin care se extrag cele n nodurile ale drumului într-un vector D . În acest algoritm, se inițializează vectorul cu nodurile cu etichetele 1 și 2 sau 2 și 1, în funcție de arcul care există [1,2], respectiv [2,1], după care se inserează în vectorul drumului și celelalte noduri, în funcție de arcele care există în graf.

PAS1. Dacă există arcul [1,2], atunci $D[1]=1$ și $D[2]=2$; altfel, $D[1]=2$ și $D[2]=1$.

PAS2. Se inițializează lungimea drumului, m , cu 2.

PAS3. Pentru nodurile cu eticheta i de la 3 până la n execută:

PAS4. Dacă există arcul $[i, 1]$, atunci poziția de inserare k a nodului i este 1 și se trece la Pasul 7; altfel, pentru a parcurge nodurile din drum se inițializează indicele j cu valoarea 1 și se trece la Pasul 5.

PAS5. Cât timp nu s-a ajuns la capătul vectorului D (indicele j nu are valoarea m) și nu s-a găsit poziția de inserare a nodului i , execută:

PAS6. Dacă există arcul între nodul cu indicele j din drum și nodul i și există și arcul între nodul i și nodul cu indicele $j+1$ din drum, atunci poziția de inserare k este $j+1$; altfel, se trece la următorul nod din drum prin incrementarea variabilei j și se revine la Pasul 6.

PAS7. Se deplasează elementele vectorului din poziția k (de inserare) spre dreapta.

PAS8. Se inserează nodul i în poziția k .

PAS9. Se incrementează lungimea m a drumului D și revine la Pasul 3.

Implementarea algoritmului. Informațiile despre graful neorientat se găsesc în fișierul *graf_t.txt*: pe prima linie numărul de noduri, apoi matricea de adiacență. Nodurile drumului elementar vor fi generate în vectorul D . Se folosesc următoarele funcții: *citeste()* pentru a citi matricea de adiacență din fișier, *generare()* pentru a genera vectorul D și *afisare()* pentru a afișa drumul elementar găsit. În funcția *generare()* se folosesc următoarele variabile locale: i – pentru eticheta nodului care se adaugă la drum, j – pentru a căuta eticheta nodului după care se inserează în drum nodul cu eticheta i , k – pentru poziția în care se inserează în vectorul drumului nodul cu eticheta i , m – pentru lungimea drumului la un moment dat (numărul de noduri adăugate la drum) și *gasit* – pentru a sădăcă s-a găsit poziția de inserare a nodului i în vector (este o variabilă logică – pentru fiecare nod i este inițializată cu valoarea *False* – 0 –, cu semnificația că nu s-a găsit încă poziția de inserare). Pentru testarea programului se folosește graful G_{37} .

```
#include<iostream.h>
int n,a[10][10],D[10];
fstream f("graf_t.txt",ios::in);
void citeste() { //se citește matricea de adiacență }
void generare()
{
    int i,j,k,m=2,gasit;
    if (a[1][2]==1){D[1]=1; D[2]=2;}
    else {D[1]=2; D[2]=1;}
    for (i=3;i<=n;i++)
    {
        if (a[i][D[1]]==1) k=1;
        else
        {
            for (j=1,gasit=0;j<=n && !gasit;j++)
                if (a[D[j]][i]==1 && a[i][D[j+1]]==1) gasit=1;
            k=j+1;
        }
        for (j=m+1;j>k;j--) D[j]=D[j-1];
        D[k]=i; m++; }
}
void afisare()
{
    for (int i=1;i<=n;i++) cout<<D[i]<<" ";
}
void main() {citeste(); generare(); afisare(); }
```



2.7.10.5. Aplicații practice

1. Graful migrației păsărilor călătoare este un graf bipartit, cele două mulțimi fiind formate din locațiile din zona rece și locațiile din zona caldă. Verificați dacă acest graf este un graf bipartit complet. În caz afirmativ, ce concluzie trageți despre migrația speciei de păsări studiate?
2. Considerați că graful din figura 39 reprezintă graful orașelor prin care trebuie să treacă voiajorul comercial. Adăugați fiecărei muchii distanța în kilometri între orașe. Scrieți un program care să rezolve problema voiajorului comercial – care să găsească traseul cu costul minim. **Indicație.** Fiecare ciclu hamiltonian găsit (nodurile depuse în stivă), va fi memorat într-o matrice cu n coloane, numărul de linii fiind egal cu numărul de cicluri găsite. Alegeți din ciclurile găsite pe cel cu costul minim.
3. **Jocul icosian** a fost inventat de William Hamilton în anul 1857. Era un dodecaedru regulat confectionat din lemn (un poliedru cu 12 fețe sub formă de pentagoane regulate) și care avea în fiecare vârf câte un cui care era marcat cu numele unui oraș (figura 44). Cu ajutorul unei sfere, care trecea prin fiecare cui o singură dată, trebuie să se găsească un traseu care să parcurgă toate orașele (să parcurgă toate muchiile dodecaedrului) cu revenire la orașul de plecare. Scrieți un program care să găsească soluții pentru acest joc, în trei variante: a) se cunoaște orașul de plecare; b) se cunosc primele două orașe prin care trebuie să treacă traseul; c) se cunosc primele trei orașe prin care trebuie să treacă traseul. Etichetele orașelor inițiale se comunică de la tastatură. **Indicație.** Dodecaedrul poate fi reprezentat cu ajutorul unui graf neorientat, în care trebuie să se găsească un circuit hamiltonian. Nodurile sunt cele 20 de vârfuri ale poliedrului (figura 45).

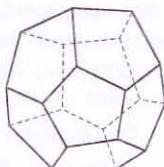


Fig. 44

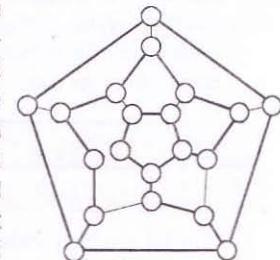


Fig. 45

4. **Jocul de șah.** Să se găsească un traseu de acoperire a tablei de șah cu una dintre piese (piesa trebuie să treacă o singură dată prin fiecare dintre cele 64 de pătrate ale tablei de șah și să revină la punctul de plecare). Piesa poate fi:
 - a. calul – se deplasează în formă de L (problema a fost studiată de Euler în anul 1759);
 - b. dama – se deplasează în oricare dintre pătratele adiacente;
 - c. nebunul – se deplasează numai în pătratele aflate pe diagonala păratului în care se găsește;
 - d. tura – se deplasează numai în pătratele aflate pe verticală sau pe orizontală față de păratul în care se găsește.**Indicație.** Tabla de șah poate fi reprezentată cu ajutorul unui graf neorientat în care trebuie să se găsească un circuit hamiltonian. Nodurile sunt cele 64 de pătrate ale tablei de șah, iar între două noduri există muchie numai dacă piesa poate să se deplaseze între cele două pătrate, conform regulilor jocului.
5. Scrieți un program care să demonstreze că problema podurilor din orașul Königsberg nu are soluții.
6. O persoană dorește să viziteze mai multe cabane, legate prin trasee turistice, astfel încât să parcurgă fiecare traseu o singură dată și să se reîntoarcă la locul de plecare. Legătura directă dintre două cabane este caracterizată prin timpul necesar parcurgerii traseului de munte (costul asociat fiecărei muchii). Scrieți un program care să găsească traseul turistic și care să determine timpul necesar parcurgerii lui.

7. **Masa rotundă a regelui Arthur.** Scrieți un program care să afișeze soluția aranjării la masa rotundă a regelui Arthur a celor $2 \times n$ cavaleri, știind că fiecare dintre cavaleri are $n-1$ dușmani și că la masă nici un cavaler nu trebuie să stea lângă dușmanul lui. **Indicație.** Relațiile dintre cavaleri pot fi reprezentate cu ajutorul unui graf neorientat în care nodurile sunt cavalerii, iar muchiile leagă numai cavaleri care nu sunt dușmani. Demonstrați că acest graf este un graf eulerian și găsiți un ciclu eulerian.
8. Scrieți un program care să găsească soluția de a desena figura 46 fără a ridica creionul de pe hârtie și fără a trece cu creionul pe o latură deja desenată. **Indicație.** Figura geometrică poate fi reprezentată printr-un graf neorientat, în care nodurile sunt vârfurile figurii, iar muchiile, laturile. Problema constă în a găsi în acest graf, care nu este eulerian, un lanț eulerian.
9. O competiție sportivă se desfășoară prin meciuri directe între doi participanți. Nu există meciuri nule și nici meciuri eliminatorii (trebuie să existe câte un meci între fiecare doi participanți). Scrieți un program care să afișeze o soluție de organizare a competiției sportive (ordinea de desfășurare a meciurilor). **Indicație.** Deoarece doi participanți nu joacă împreună decât un singur meci, meciul va fi definit prin relația nesimetrică sportivul x joacă cu sportivul y (adică, dacă sportivul x joacă cu sportivul y , atunci sportivul y nu mai joacă cu sportivul x). Graful campionatului este un graf orientat, în care nodurile reprezintă sportivi, iar muchiile, meciurile. Prin modul de organizare a competiției, el este un graf turneu.

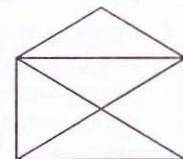


Fig. 46

Evaluare

Adevărat sau Fals:

1. Nodul x este incident cu nodul y dacă formează împreună o muchie.
2. Gradul intern al unui nod dintr-un graf orientat este egal cu numărul de arce care ies din nod.
3. Într-un graf orientat, mulțimea predecesorilor unui nod este formată din mulțimea nodurilor de la care ajung arce care intră în nod.
4. Într-un graf orientat, suma gradelor interne și a gradelor externe ale tuturor nodurilor este egală cu dublul numărului de muchii.
5. Matricea de incidentă a unui graf orientat este o matrice binară.
6. Într-un graf orientat, numărul de vecini ai unui nod este egal cu suma dintre gradul intern și gradul extern ale nodului.
7. Lista de adiacență a unui graf orientat este formată din listele vecinilor fiecărui nod din graf.
8. Graful nul conține numai noduri izolate.
9. Un graf neorientat cu 5 noduri poate conține maxim 10 muchii.
10. Un graf orientat cu 4 noduri poate conține maxim 12 arce.
11. Un subgraf se obține prin eliminarea unor muchii (arce) din graf.
12. Într-un lanț elementar fiecare muchie se parcurge o singură dată.
13. Într-un ciclu elementar o muchie poate fi parcursă de mai multe ori.
14. Un drum este un lanț într-un graf orientat.
15. În matricea drumurilor elementul $a[i][j]$ are valoarea 1 numai dacă există un drum de la nodul i la nodul j și un drum de la nodul j la nodul i .
16. O componentă conexă a unui graf este un graf parțial conex al acestuia.
17. Prin parcurgerea în lățime a unui graf se pot determina componentele conexe ale grafului.

18. Un graf conex cu n noduri și $n-1$ muchii conține cel puțin un ciclu.
19. Un graf tare conex este un graf orientat conex.
20. Un graf bipartit complet este un graf complet care este bipartit.
21. Cu patru noduri se pot genera 14 grafuri bipartite complete.
22. Un graf hamiltonian poate conține noduri izolate.
23. Dacă într-un graf cu n noduri ($n \leq 3$) gradul fiecărui nod este mai mic decât $n/2$, atunci graful nu este hamiltonian.
24. Lanțul eulerian este un lanț elementar care parcurge toate muchiile grafului.
25. Un graf eulerian poate conține p componente conexe, cu condiția ca $p-1$ componente conexe să fie formate numai din noduri izolate.
26. Într-un graf turneu cu n noduri, suma dintre gradul intern și gradul extern ale oricărui nod este n .
27. Numărul de arce dintr-un graf turneu cu n noduri este $n \times (n-1)/2$.

Alegeți:

1. Determinați numărul total de grafuri neorientate distințe cu 3 noduri. Două grafuri se consideră distințe dacă matricea lor de adiacență diferă.

a. 4	b. 7	c. 64	d. 8
------	------	-------	------

(Bacalaureat – Sesiunea specială 2003)

2. Numărul de grafuri orientate care se pot construi cu 3 noduri este:

a. 8	b. 9	c. 64	d. 16
------	------	-------	-------

3. Numărul maxim de arce într-un graf orientat cu 5 noduri este:

a. 5	b. 10	c. 20	d. 25
------	-------	-------	-------

Următorii 12 itemi se referă la graful neorientat cu 8 noduri și având muchiile [1,2], [1,3],

[1,4], [1,5], [2,3], [3,4], [3,7], [4,7], [4,5].

4. Numărul de noduri care au gradul maxim este:

a. 1	b. 2	c. 4	d. 3
------	------	------	------

5. Numărul de noduri izolate este:

a. 1	b. 2	c. 0	d. 3
------	------	------	------

6. Numărul de elemente de 0 din matricea de adiacență este:

a. 9	b. 18	c. 20	d. 46
------	-------	-------	-------

7. Numărul de elemente de 0 din matricea de incidentă este:

a. 18	b. 9	c. 36	d. 54
-------	------	-------	-------

8. Suma elementelor de pe linia 1 din matricea de incidentă este:

a. 8	b. 4	c. 2	d. 6
------	------	------	------

9. Numărul de cicluri elementare de lungime 4 este:

a. 4	b. 3	c. 6	d. 5
------	------	------	------

10. Numărul de lanțuri elementare de lungime 4 este:

a. 16	b. 8	c. 10	d. 22
-------	------	-------	-------

11. Numărul de lanțuri elementare între nodul 1 și nodul 7 este:

a. 6	b. 3	c. 4	d. 8
------	------	------	------

12. Numărul de subgrafuri cu 3 noduri care se pot obține din graf este:

a. 10	b. 3	c. 20	d. 56
-------	------	-------	-------

13. Numărul de grafuri parțiale cu 7 muchii care se pot obține din graf este:

a. 72	b. 18	c. 36	d. 9
-------	-------	-------	------

14. Numărul minim de muchii care se pot elimina din componenta conexă {1,2,3,4,5,7} a grafului pentru a se obține un graf parțial neconex este:

a. 1	b. 4	c. 3	d. 2
------	------	------	------

15. Numărul maxim de muchii care se pot elimina din componenta conexă {1,2,3,4,5,7} a grafului pentru a se obține un graf parțial conex este:

- a. 5 b. 4 c. 3 d. 2

Următorii 11 itemi se referă la graful orientat cu 6 noduri și cu arcele [1,2], [1,3], [1,4], [1,5], [2,3], [3,1], [3,4], [3,6], [4,1], [4,5], [4,6], [6,4].

16. Numărul de noduri care au gradul intern egal cu gradul extern este:

- a. 1 b. 2 c. 4 d. 3

17. Numărul de elemente de 0 din matricea de adiacență este:

- a. 9 b. 18 c. 26 d. 24

18. Numărul de elemente de 1 din matricea de incidentă este:

- a. 22 b. 11 c. 6 d. 8

19. Suma elementelor de pe linia 4 din matricea de incidentă este:

- a. 8 b. 4 c. 0 d. 6

20. Suma gradelor externe și a gradelor interne ale tuturor nodurilor este:

- a. 12 b. 24 c. 20 d. 14

21. Numărul de cicluri elementare de lungime 4 este:

- a. 4 b. 3 c. 0 d. 6

22. Numărul de lanțuri elementare de lungime 4 este:

- a. 16 b. 8 c. 10 d. 22

23. Numărul de circuite elementare de lungime 4 este:

- a. 2 b. 4 c. 3 d. 5

24. Numărul de lanțuri elementare între nodul 1 și nodul 6 este:

- a. 6 b. 3 c. 4 d. 8

25. Numărul de drumuri elementare între nodul 1 și nodul 6 este:

- a. 6 b. 3 c. 4 d. 5

26. Numărul de componente tare conexe este:

- a. 1 b. 3 c. 2 d. 4

27. Se consideră graful neorientat dat prin matricea de adiacență alăturată. În graf, nodurile sunt numerotate de la 1 la 4, corespunzător liniilor tabloului. Să se determine câte perechi de noduri neadiacente există în graf. Se consideră perechile neordonate (perechea 1–2 este aceeași cu perechea 2–1).

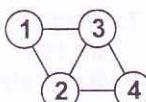
- a. 1 b. 2 c. 3 d. 4

(Bacalaureat – Sesiunea august 2003)

28. Se consideră graful neorientat din figura alăturată. Să se determine care dintre următoarele liste reprezintă lista de adiacență a nodului 3:

- a. 1 2 4 b. 3 1 2 4 c. 2 4 d. 1 3 4

(Bacalaureat – Sesiunea iunie-iulie 2003)



29. Se consideră un graf orientat având matricea de adiacență alăturată. Stabilități care dintre nodurile grafului au gradul intern egal cu gradul extern.

- a. 3 b. 1 c. 0 d. 2

(Bacalaureat – Simulare 2003)

30. Într-un graf neorientat cu n noduri numărul maxim de muchii ce pot exista este:

- a. $n \times (n+1)/2$ b. $n \times (n-1)/2$ c. $n \times (n-1)$ d. n^2

(Bacalaureat – Sesiunea iunie-iulie 2003)

31. Numărul maxim de muchii într-un graf neorientat cu 8 noduri care nu conține niciun ciclu este:

- a. 4 b. 8 c. 27 d. 7

(Bacalaureat – Simulare 2005)

32. Știind că un graf neorientat fără noduri izolate are 7 muchii și 8 noduri, stabiliți numărul maxim de componente conexe din care poate fi format acesta.

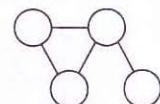
a. 1

b. 2

c. 3

d. 4

(Bacalaureat – Sesiunea august 2005)



33. Se consideră graful din figura alăturată. Determinați matricea de adiacență corespunzătoare:

0	1	1	1
1	0	0	0
1	0	1	0
1	0	1	0

0	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0

1	1	0	1
1	1	0	1
0	0	1	1
1	1	1	1

0	1	0	1
1	0	0	1
0	0	0	1
1	1	1	0

(Bacalaureat – Simulare 2003)

34. Se consideră graful neorientat din figura alăturată. Să se determine eticheta nodului care are lista de adiacență 2 3 5.

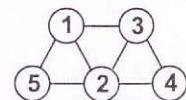
a. 5

b. 4

c. 3

d. 1

(Bacalaureat – Sesiunea iunie-iulie 2003)



35. Se consideră un graf orientat dat prin liste de adiacență: 1) 2 4 2) 4 3) 1 5 4) 4 2 3 5) 2 3 4. Stabiliți numărul de noduri care au gradul intern egal cu gradul extern.

a. 3

b. 1

c. 3

d. 2

(Bacalaureat – Sesiunea iunie-iulie 2003)

36. Numărul minim de noduri dintr-un graf neorientat cu 12 muchii, fără noduri izolate, graf format din exact 3 componente conexe:

a. 7

b. 8

c. 9

d. 10

(Bacalaureat – Sesiunea specială 2005)

37. Numărul minim de componente conexe ale unui graf neorientat cu 9 muchii și 12 noduri este::

a. 2

b. 3

c. 4

d. 5

(Bacalaureat – Sesiunea iunie-iulie 2005)

38. Care este numărul minim de muchii care pot fi plasate într-un graf neorientat cu 53 de noduri, astfel încât să nu existe nici un nod izolat?

a. 27

b. 26

c. 53

d. 52

39. Se consideră un graf neorientat G cu 8 noduri și 16 muchii. Numărul de noduri izolate din G este:

a. exact 1

b. exact 0

c. cel mult 1

d. cel mult 2

(Bacalaureat – Sesiunea iunie-iulie 2003)



40. Stabiliți care dintre următoarele matrice de adiacență corespunde grafului din figura alăturată.

0	1	1	0
1	0	1	0
1	1	0	1
0	0	1	0

0	1	0	1	0
1	0	0	1	0
0	0	0	0	0
1	1	0	0	0
0	0	0	1	0

0	0	0	1	1
0	0	0	0	1
0	0	0	0	0
1	0	0	0	1
1	1	0	1	0

0	0	0	1
0	0	1	1
0	1	0	1
1	1	1	0

(Bacalaureat – Sesiunea august 2003)

41. Într-un graf neorientat cu 20 de noduri numerotate de la 1 la 20 există muchii numai între perechile de noduri i și j cu proprietatea $\text{abs}(i-j) > 1$. Numărul de valori egale cu 1 din matricea de adiacență corespunzătoare grafului este:

a. 190

b. 362

c. 340

d. 171

(Bacalaureat – Sesiunea iunie-iulie 2003)

42. Care este matricea de adiacență a unui graf neorientat cu 4 noduri, 2 muchii și cel puțin un nod izolat?

a)	<table border="1"><tr><td>0</td><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	1	0	1	0	0	0	1	0	1	0	0	0	0	0
0	1	1	0														
1	0	0	0														
1	0	1	0														
0	0	0	0														
b)	<table border="1"><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	0	0	1	0	0	1	1	1	0							
0	0	1															
0	0	1															
1	1	0															

b)	<table border="1"><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	0	0	1	0	0	1	1	1	0
0	0	1								
0	0	1								
1	1	0								

c)	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	0	1	1	0	0	0	0	0	0	0	1	0	0	0
0	1	0	1														
1	0	0	0														
0	0	0	0														
1	0	0	0														

d)	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td></tr></table>	0	0	0	0	0	0	1	1	0	1	0	1	0	1	1	0
0	0	0	0														
0	0	1	1														
0	1	0	1														
0	1	1	0														

(Bacalaureat – Simulare 2004)

43. Se consideră un graf orientat cu 6 noduri etichetate cu numere de la 1 la 6 și 6 arce astfel încât există un arc de la fiecare nod cu eticheta i către un nod cu eticheta $i \times 2$ dacă există un astfel de nod sau către nodul cu eticheta $i-1$ în caz contrar. Care este lungimea maximă a unui drum în graf?

a. 4

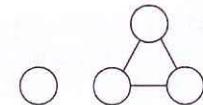
b. 3

c. ∞

d. 2

(Bacalaureat – Simulare 2003)

44. Stabilită care dintre următoarele variante este matricea de adiacență a unui subgraf al grafului din figura alăturată:



a)	<table border="1"><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	0
0	0	0								
0	0	0								
0	0	0								
b)	<table border="1"><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr></table>	0	1	0	1	0	1	0	1	1
0	1	0								
1	0	1								
0	1	1								

b)	<table border="1"><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr></table>	0	1	0	1	0	1	0	1	1
0	1	0								
1	0	1								
0	1	1								

c)	<table border="1"><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr></table>	0	1	1	1	0	1	1	0	1
0	1	1								
1	0	1								
1	0	1								

d)	<table border="1"><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	0	1	1	1	0	1	1	1	0
0	1	1								
1	0	1								
1	1	0								

(Bacalaureat – Sesiunea specială 2003)

45. În matricea de adiacență asociată grafului orientat din figura alăturată nodurile sunt numerotate de la 1 la 4 corespunzător liniilor matricei.

Numărul de elemente egale cu 1 aflate deasupra diagonalei principale depășește numărul de elemente egale cu 1 aflate sub diagonala principală cu

a. 2

b. 0

c. 3

d. 1

(Bacalaureat – Sesiunea iunie-iulie 2004)

46. Se consideră un graf orientat cu nodurile numerotate cu 1, 2, 3, 4, 5, 6 corespunzător liniilor matricei de adiacență alăturate. Stabilită dacă nodurile 1 și 3:

- a. aparțin aceleiași componente conexe
c. sunt conectate printr-un lanț

- b. sunt noduri izolate
d. sunt noduri adiacente

(Bacalaureat – Simulare 2003)

47. Care dintre următoarele secvențe de noduri reprezintă un lanț în graful neorientat dat prin matricea de adiacență alăturată știind că nodurile sunt numerotate de la 1 la 5 corespunzător liniilor tabloului?

a. 3 4 1 2 5 b. 1 5 3 4 2 4

c. 4 1 2 4 5

d. 2 1 3 4

(Bacalaureat – Sesiunea iunie-iulie 2004)

48. Se consideră graful neorientat dat prin matricea de adiacență alăturată. În graf, nodurile sunt numerotate de la 1 la 4, corespunzător liniilor tabloului. Să se determine lungimea maximă a unui lanț ce unește nodurile 1 și 3:

a. 1

b. 2

c. 3

d. 4

(Bacalaureat – Sesiunea iunie-iulie 2003)

49. Care dintre următoarele secvențe reprezintă un drum în graful orientat dat prin matricea de adiacență alăturată, știind că nodurile sunt numerotate de la 1 la 4 corespunzător liniilor și coloanelor tabloului?

a. 3 4 3 2 4 b. 3 1 2 3

c. 2 1 3 4

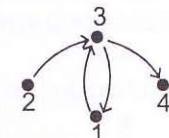
d. 4 3 1 2

(Bacalaureat – Sesiunea specială 2004)

50. Cel mai lung drum elementar în graful orientat din figura alăturată are lungimea

- a. 2 b. 4 c. ∞

(Bacalaureat – Sesiunea iunie-iulie 2004)



51. Se consideră un graf neorientat cu 7 noduri și 3 muchii. Numărul de componente conexe din care poate fi format graful este:

- a. exact 4 b. 4 sau 5 c. 3 sau 4 d. cel puțin 5

(Bacalaureat – Simulare 2003)

52. Dacă într-un graf neorientat conex cu n noduri, lista de adiacență a fiecărui nod este formată din exact două elemente, atunci în graful respectiv există:

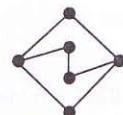
- a. cel puțin $n/2$ cicluri b. exact $n/2$ cicluri
c. exact un ciclu d. cel puțin două cicluri

(Bacalaureat – Sesiunea specială 2003)

53. Numărul minim de muchii ce se pot alege pentru a fi eliminate din graful neorientat din figura alăturată astfel încât acesta să devină neconex este:

- a. 2 b. 4 c. 3 d. 1

(Bacalaureat – Sesiunea iunie-iulie 2003)



54. Numărul de grafuri conexe aciclice care se pot obține cu 4 noduri este:

- a. 2 b. 4 c. 28 d. 24

55. Se consideră un graf orientat tare conex. Stabiliți numărul circuitelor care conțin toate nodurile grafului:

- a. exact unul b. cel mult unul c. nici unul d. cel puțin unul

(Bacalaureat – Sesiunea iunie-iulie 2003)

56. Gradul maxim al unui nod ce se poate obține într-un graf neorientat conex cu n noduri și $n-1$ muchii este (s-a notat cu $[n/2]$ câtul împărțirii întregi a lui n la 2):

- a. $n-1$ b. $[n/2]$ c. n d. 2

(Bacalaureat – Sesiunea iunie-iulie 2003) 0 0 1 0 0
0 0 0 1 1

57. Se consideră graful neorientat având nodurile notate cu 1, 2, 3, 4, 5 corespunzător liniilor matricei de adiacență alăturate. Stabiliți care dintre următoarele propoziții este adevărată.

1 0 0 0 0
0 1 0 0 1
0 1 0 1 0

- a. graful este conex b. orice nouă muchie s-ar adăuga graful devine conex
c. graful este aciclic d. orice muchie s-ar elimina graful devine aciclic

(Bacalaureat – Simulare 2003)

58. În graful neorientat dat prin matricea de adiacență alăturată numărul de componente conexe este:

- a. 0 b. 2 c. 3 d. 1

(Bacalaureat – Sesiunea iunie-iulie 2003) 0 0 1 0
0 0 0 0

59. Stabiliți care dintre următoarele matrice de adiacență corespunde unui graf conex aciclic:

- | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a) | <table border="1"> <tr><td>0</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>0</td></tr> </table> | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | b) | <table border="1"> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> </table> | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | c) | <table border="1"> <tr><td>0</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>0</td></tr> </table> | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | d) | <table border="1"> <tr><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td><td>0</td></tr> </table> | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 0 | 1 | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 1 | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 1 | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 0 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 1 | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 1 | 0 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 0 | 1 | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 1 | 0 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 0 | 1 | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 1 | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 0 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 0 | 0 | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 1 | 0 | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

(Bacalaureat – Sesiunea august 2003)

60. Câte grafuri hamiltoniene se pot crea cu 4 noduri?

- a. 1 b. 2 c. 4 d. 5

61. Câte grafuri hamiltoniene se pot crea cu 4 noduri și 5 muchii?

- a. 1 b. 2 c. 4 d. 5

62. Câte grafuri euleriene se pot crea cu 4 noduri?

- a. 1 b. 2 c. 4 d. 5

63. Numărul de cicluri hamiltoniene dintr-un graf complet cu 4 noduri este:

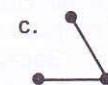
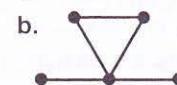
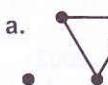
- a. 1 b. 3 c. 4 d. 6

64. Determinați numărul total de grafuri neorientate conexe cu 3 noduri, două grafuri fiind considerate distincte dacă și numai dacă matricele lor de adiacență diferă prin cel puțin un element.

- a. 7 b. 8 c. 4 d. 2

(Bacalaureat – Sesiunea august 2003)

65. Care dintre următoarele figuri reprezintă un graf neorientat conex?



(Bacalaureat – Sesiunea august 2004)

66. Considerând un graf neorientat având nodurile 1,2,3,4,5, 6,7,8,9 corespunzător liniilor matricei de adiacență alăturate, stabiliți care dintre următoarele propoziții este adevărată:

- | | | |
|--------------------------|-----------------------------|-------------------|
| a. este un graf bipartit | b. este un graf conex | 0 1 1 0 0 0 0 0 0 |
| c. este un graf eulerian | d. este un graf hamiltonian | 1 0 1 0 0 1 0 0 0 |

67. Câte muchii are graful bipartit complet $G=(X,U)$, cu 11 noduri și cu mulțimea nodurilor A și B definite astfel: $A \cup B = X$, $A \cap B = \emptyset$ și $\text{card}(A) = 4$?

- | | | | | |
|------|-------|-------|-------|-------------------|
| a. 4 | b. 10 | c. 16 | d. 28 | 1 1 0 1 1 0 0 0 0 |
|------|-------|-------|-------|-------------------|

68. Care dintre următoarele afirmații sunt adevărate?

- a. un graf hamiltonian este un graf care conține un ciclu elementar care trece prin toate nodurile;
- b. un graf eulerian este un graf care conține un ciclu simplu care trece prin toate muchiile;
- c. un graf fără noduri izolate este eulerian dacă este conex și gradul fiecărui nod este par;
- d. un graf complet este hamiltonian.

69. Stabiliți care dintre următoarele proprietăți ale unui graf turneu sunt adevărate:

- a. graful turneu trebuie să fie un graf hamiltonian;
- b. graful turneu trebuie să fie un graf eulerian;
- c. graful turneu este un graf orientat în care, între oricare două noduri, există un arc și numai unul, care le leagă;
- d. graful turneu este un graf conex ciclic.

70. Stabiliți cu care dintre următoarele valori este egală suma elementelor matricei de adiacență a grafului turneu:

- a. $n \times (n-1)$ b. $n \times (n-1)/2$ c. n d. $n-1$

71. Stabiliți care dintre următoarele proprietăți ale matricei de adiacență a grafului turneu este adevărată:

- a. suma elementelor din linia i și coloana j este egală cu n;
- b. suma elementelor din linia i și coloana j este egală cu $n-1$;
- c. suma elementelor din linia i și coloana j este egală cu $n-1$, oricare ar fi i și j;
- d. suma elementelor din linia i și coloana j este egală cu n, oricare ar fi i și j.

72. Stabiliți care dintre următoarele egalități între elementele matricei de adiacență a grafului turneu sunt adevărate:
- $a[i][j] = 1 - a[j][i]$, oricare ar fi $i \neq j$;
 - $a[i][j] + a[j][i] = 1$, oricare ar fi $i \neq j$;
 - $a[i][j] = 1 - a[j][i]$, oricare ar fi $i \neq j$;
 - $a[i][j] = 1 - a[j][i]$, oricare ar fi $i \neq j$;
73. Un graf orientat având nodurile 1,2,3,4,5 corespunzător liniilor matricei de adiacență alăturate reprezintă un graf turneu pentru un concurs hipic, definit la Studiul de caz, în care nodurile sunt obstacolele. Traseul pe care poate să-l urmeze un călăreț, începând cu obstacolul 1, este:
- | | | | |
|--------------|----------------|--------------|--------------|
| a. 1,3,2,4,5 | b. 1,3,4,2,3,5 | c. 1,3,5,4,2 | d. 1,3,4,5,2 |
|--------------|----------------|--------------|--------------|
74. Se consideră graful turneu care corespunde matricei de adiacență de la problema anterioară. Un călăreț care începe cu obstacolul 1 poate alege din n variante de trasee, unde n are valoarea:
- | | | | |
|------|------|------|------|
| a. 2 | b. 3 | c. 4 | d. 5 |
|------|------|------|------|
75. Considerăm un graf neorientat având nodurile 1,2,3,4,5 corespunzător liniilor matricei de adiacență alăturate. Câte muchii trebuie adăugate pentru ca graful să fie un graf eulerian și hamiltonian?
- | | | | |
|------|------|------|--------------|
| a. 1 | b. 2 | c. 3 | d. imposibil |
|------|------|------|--------------|

Miniproiecte:

- Rețeaua de străzi a unui oraș este formată din intersecții și străzi pe care se poate circula în ambele sensuri sau într-un singur sens. Fiecare stradă are o lungime măsurată în metri. Două automobile pleacă la același moment, unul din intersecția A și altul din intersecția B, și trebuie să ajungă fiecare în intersecția C, respectiv intersecția D. Cele două automobile merg cu viteza medie v_1 , respectiv v_2 . Ambii automobilisti aleg traseul între cele două intersecții astfel încât distanța parcursă să fie minimă. Informațiile despre rețeaua de străzi (numărul de intersecții, traficul auto între intersecții și lungimea fiecărei străzi) se citesc dintr-un fișier text. Se citesc de la tastatură etichetele celor patru intersecții și vitezele automobilelor. Scrieți o aplicație care să furnizeze următoarele informații:
 - Care dintre automobilisti ajunge mai repede la destinație?
 - Dacă există intersecții prin care trec ambii automobilisti, să se afișeze aceste intersecții.
 - Dacă automobilistii se întâlnesc într-o intersecție, să se precizeze această intersecție.
 - Dacă, după ce au ajuns la destinație, cei doi automobilisti pornesc unul către celălalt, care este cel mai scurt drum pe care trebuie să meargă, și în ce intersecție își propun să se întâlnească, astfel încât timpul de așteptare în intersecție a automobilistului care ajunge primul să fie minim?
- O rețea de calculatoare are n servere. La calculatoarele server pot fi conectate mai multe calculatoare client. Două calculatoare server pot comunica prin legături directe sau prin intermediul altor servere. Informațiile despre topologia rețelei (numărul de severe și legăturile directe dintre servere) se citesc dintr-un fișier text. Scrieți o aplicație care să furnizeze următoarele informații:
 - Care este drumul cel mai scurt pe care un mesaj transmis de un client al serverului x să ajungă la un client al serverului y (etichetele severelor se citesc de la tastatură)?
 - Să se precizeze care sunt calculatoarele server care, dacă se defectează, fac ca rețeaua să nu mai fie funcțională (unele severe din rețea nu mai pot comunica între ele).
 - Să se precizeze care este serverul cel mai solicitat. Serverul cel mai solicitat este serverul prin care este posibil să treacă, la un moment dat, cele mai multe mesaje (serverul prin care trec cele mai multe lanțuri).

- d. Să se reproiecteze rețeaua prin adăugarea unui număr minim de legături directe între servere, astfel încât oricare dintre servere s-ar defecta, rețeaua să rămână funcțională (cu excepția clientilor serverului defect, restul clientilor trebuie să aibă acces la resursele rețelei).
3. Într-o fabrică se realizează un tip de produs care este format din mai multe subansambluri. Fiecare subansamblu este produs de o secție. Un subansamblu poate fi, la rândul său, compus din alte subansambluri. Există o secție în care se asamblează produsul final și secțiile care, pentru a produce propriul subansamblu, nu au nevoie de subansambluri produse în secțiile fabricii. Secțiile fabricii și relațiile dintre secții pot fi reprezentate printr-un graf orientat, astfel: secțiile sunt nodurile grafului, iar arcul $[x,y]$ înseamnă că subansamblul produs de secția x este folosit de secția y . Desenați o organigramă ipotecă a secțiilor fabricii. Informațiile despre organigramă fabricii se citesc dintr-un fișier text: de pe prima linie n – numărul de secții și m – numărul de arce ale grafului, de pe următorul rând un sir de n numere întregi care reprezintă stocul de subansambluri realizate în fiecare secție, de pe următorul rând un sir de n numere întregi care reprezintă numărul de angajați din fiecare secție și apoi, de pe următoarele m rânduri, câte trei valori numerice întregi care reprezintă etichetele nodurilor terminale ale unui arc (x și y) și numărul de subansambluri pe care trebuie să le furnizeze secția x secției y , pentru a se putea obține o unitate din produsul final. Scrieți o aplicație care să furnizeze următoarele informații:
- Să verifice dacă se poate realiza o unitate din produsul final folosind subansamblurile care există pe stoc în secțiile fabricii. Dacă nu se poate realiza o unitate din produsul final, să se precizeze secțiile care sunt vinovate că nu au produs suficiente subansambluri.
 - În fabrică, se închid două dintre secțiile care realizează subansambluri (subansamblurile se vor cumpăra de la alți producători). Etichetele nodurilor corespunzătoare acestor secții se citesc de la tastatură. Verificați dacă, prin închiderea acestor secții, alte secții ale fabricii nu devin inutile. O secție devine inutilă atunci când prin închiderea unei alte secții, subansamblurile produse de ea nu mai sunt necesare. Dacă apar secții inutile, în urma închiderii celor două secții inițiale, afișați eticheta corespunzătoare secției și închideți și aceste secții. Reorganizați fabrica prin eliminare de secții – până când nu vor mai exista secții inutile. Afișați numărul de angajați care trebuie disponibilizați prin închiderea secțiilor. Afișați organigramă fabricii după reorganizare, prin etichetele secțiilor care au mai rămas, și arcele dintre acestea. (Indicație. Generați subgraful inițial, prin eliminarea nodurilor corespunzătoare secțiilor care se închid. Generați apoi subgrafuri ale ultimului subgraf obținut prin eliminarea nodurilor care au gradul extern egal cu 0. Subgraful final obținut va corespunde noii organigrame. Când eliminați un nod, adunați la numărul angajaților disponibilizați angajații din secția corespunzătoare nodului).
4. Pentru realizarea unui proiect trebuie executate mai multe activități (analiza, elaborarea modulelor de program, testarea, elaborarea documentației etc.). Unele activități sunt condiționate de execuția altora (de exemplu, nu se poate scrie un modul de program fără să se fi făcut analiza aplicației), alte activități se pot desfășura independent unele de altele (se pot scrie module de program sau se pot testa unele dintre module, independent unele de altele). Fiecare activitate are un timp maxim de execuție. Să se reprezinte sub forma unui graf orientat activitățile de realizare a fiecăruiuia dintre miniproiectele anterioare și să se determine timpul maxim de execuție al fiecăruiuia dintre ele.

2.8. Arborele

2.8.1. Arborele liber

2.8.1.1. Definiția arborelui liber

Se numește **arbore liber** A un graf neorientat conex și fără cicluri.

Observație. De obicei se omite adjectivul „liber”, referirea la un graf conex aciclic făcându-se numai cu numele arbore.

Se numește **subarbore** al arborelui A=(X,U), orice arbore S=(Y,V) care are proprietatea: $Y \subseteq X$ și $V \subseteq U$.

Exemplu – figura 47. Graful G_{38} cu 8 noduri este un arbore (un graf neorientat conex și aciclic), iar graful G_{39} este un subarbore al acestuia.

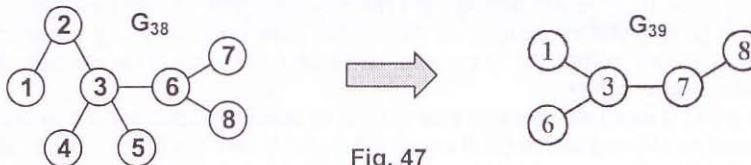


Fig. 47

2.8.1.2. Proprietățile arborilor liberi

Teorema 21

Următoarele definiții sunt echivalente pentru un graf G cu n noduri și m muchii:

- (1) G este un **arbore**.
- (2) G este un **graf aciclic** cu $n-1$ muchii.
- (3) G este un **graf conex** cu $n-1$ muchii.
- (4) G este un **graf fără cicluri maximal** (dacă în graful fără cicluri G unim două noduri oarecare neadiacente printr-o muchie, graful obținut conține un ciclu).
- (5) G este un **graf conex minimal** (dacă în graful conex G suprimăm o muchie oarecare, graful obținut nu mai este conex).
- (6) **Orice pereche de noduri este legată printr-un lanț și numai unul.**

Demonstrație. Este suficient să se demonstreze implicațiile $(1) \Rightarrow (2)$, $(2) \Rightarrow (3)$, $(3) \Rightarrow (4)$, $(4) \Rightarrow (5)$, $(5) \Rightarrow (6)$ și $(6) \Rightarrow (1)$. Prin tranzitivitatea acestor implicații rezultă implicațiile $(2) \Rightarrow (1)$, $(3) \Rightarrow (2)$, $(4) \Rightarrow (3)$, $(5) \Rightarrow (4)$, $(6) \Rightarrow (5)$ și $(1) \Rightarrow (6)$. Din aceste implicații rezultă că $(1) \Leftrightarrow (2)$, $(2) \Leftrightarrow (3)$, $(3) \Leftrightarrow (4)$, $(4) \Leftrightarrow (5)$, $(5) \Leftrightarrow (6)$ și $(6) \Leftrightarrow (1)$. Din tranzitivitatea relației de echivalentă rezultă oricare două din cele 6 propoziții sunt echivalente.

(1) \Rightarrow (2). Ipoteza: Graful G este conex și aciclic – din definiția arborelui (1).

Concluzie: Graful G este aciclic și are $n-1$ muchii (2).

Proprietatea că graful G este aciclic este comună ipotezei și concluziei. Trebuie demonstrat doar că un graf conex aciclic are $n-1$ muchii. Dacă G este conex aciclic, nu trebuie eliminată nici o muchie pentru a se obține un graf parțial conex aciclic. Cum numărul de muchii care trebuie eliminate dintr-un graf conex pentru a obține un graf parțial conex aciclic este egal cu $m-n+1$, înseamnă că în acest caz, $m-n+1=0$. Rezultă că $m=n-1$.

(2) \Rightarrow (3). Ipoteza: Graful G este aciclic și are $n-1$ muchii – din definiția arborelui (2).

Concluzie: Graful G este conex și are $n-1$ muchii (3).

Proprietatea că graful G are $n-1$ muchii este comună ipotezei și concluziei. Trebuie demonstrat doar că un graf cu $n-1$ muchii fiind aciclic este și conex. Se știe că într-un graf cu p componente conexe, numărul de muchii care trebuie eliminate pentru a obține un graf parțial aciclic este egal cu $m-n+p$.

Graful G este aciclic ($m-n+p=0$) și are $n-1$ muchii ($m=n-1$ și $(n-1)-n+p=0$). Rezultă că $p=1$ (graful are o singură componentă conexă, deci este conex).

(3) \Rightarrow (4). Ipoteza: Graful G este conex și are $n-1$ muchii (3).

Concluzie: Graful G este aciclic maximal (4).

G fiind conex, numărul de componente conexe p este egal cu 1. Numărul de muchii m ale grafului G este egal cu $n-1$. Rezultă că numărul de muchii care trebuie eliminate din graful G ca să se obțină un graf parțial aciclic este egal cu: $m-n+p = (n-1)+n+1 = 0$, adică nici o muchie. Rezultă că graful G este aciclic. Graful este maximal pentru această proprietate, deoarece fiind conex, orice muchie $[x_i, x_j]$ care se va adăuga va forma un ciclu cu lanțul $L(x_i, x_j)$ – existența acestui lanț rezultă din conexitatea grafului G.

(4) \Rightarrow (5). Ipoteza: Graful G este aciclic maximal (4).

Concluzie: Graful G este conex minimal (5).

a. Presupunem că graful G nu este conex. El are cel puțin două componente conexe: C_1 și C_2 . Înseamnă că există două noduri $x \in C_1$ și $y \in C_2$ pe care le putem lega cu muchia $[x, y]$. Dar, din ipoteză, rezultă că orice muchie am adăuga la graf, el va conține un ciclu. Rezultă că prin adăugarea muchiei $[x, y]$ graful va conține un ciclu și că, între nodurile x și y există deja un lanț și ele aparțin aceleiași componente conexe, ceea ce contrazice presupunerea făcută. Înseamnă că graful G este conex.

b. Presupunem că graful G care este conex nu este minimal cu această proprietate. Înseamnă că prin eliminarea unei muchii se obține graful parțial H care are n noduri și $m-1$ muchii, este conex și aciclic. Graful G fiind conex și aciclic, înseamnă că numărul de muchii care trebuie eliminate pentru a obține un graf parțial conex aciclic este egal cu 0, adică $m-n+1=0$ și $m=n-1$. Numărul de muchii ale grafului H este egal cu $m-1=n-2$. Fiind aciclic înseamnă că numărul de muchii care trebuie eliminate pentru a obține un graf parțial aciclic este egal cu 0, adică $(n-2)-n+p=0$. Rezultă că $p=2$, adică graful parțial H conține două componente conexe, ceea ce contrazice presupunerea că el este conex.

(5) \Rightarrow (6). Ipoteza: Graful G este conex minimal (5).

Concluzie: Orice pereche de noduri este legată printr-un lanț și numai unul (6).

Graful G fiind conex, înseamnă că există cel puțin un lanț care leagă oricare două noduri x și y. Presupunem că există cel puțin două lanțuri între nodurile x și y: L_1 și L_2 . Înseamnă că, suprimând o muchie din lanțul al doilea, graful rămâne conex, deoarece nodurile x și y vor fi legate prin lanțul L_1 , ceea ce contrazice ipoteza că graful G este conex minimal. Rezultă că cele două noduri x și y nu sunt legate decât printr-un singur lanț.

(6) \Rightarrow (5). Ipoteza: Orice pereche de noduri este legată printr-un lanț și numai unul (6).

Concluzie: Graful G este conex și aciclic – din definiția arborelui (1).

Deoarece în graful G, orice pereche de noduri x și y este legată printr-un lanț, înseamnă că graful G este conex. Presupunem că graful G conține cel puțin un ciclu. Considerând două noduri oarecare x și y care aparțin acestui ciclu, înseamnă că între cele două noduri există două lanțuri diferite, ceea ce contrazice ipoteza. Rezultă că graful G este aciclic.

Propoziția 14

Orice arbore cu n noduri are $n-1$ muchii.

Demonstrație. Arborele fiind un graf conex minimal, înseamnă că are $n-1$ muchii.

Propoziția 15. Orice arbore cu $n \geq 2$ noduri conține cel puțin două noduri terminale.

Demonstrație – prin reducere la absurd. Presupunem că nu există decât un singur nod terminal x. Oricare alt nod din arbore are cel puțin gradul 2. Alegem din arbore, dintre lanțurile elementare care au o extremitate în nodul x, lanțul de lungime maximă: $L(x, y)$. Nodul y având gradul mai mare decât 1, înseamnă că mai există, în afara nodului care îl precede în lanț, cel puțin un nod z care este adiacent cu el. Lanțul fiind elementar, înseamnă că nodul y nu există în lanț decât la extremitatea lui, iar muchia $[y, z]$ nu face parte din lanț și putem adăuga această muchie la lanț. Lanțul fiind de lungime maximă, înseamnă că nodul z aparține

lanțului și prin adăugarea acestei muchii la lanț se închide un ciclu, ceea ce contrazice definiția arborelui (graf aciclic).


Temă

- Precizați dacă este arborele graful $G_{40}(X_{40}, U_{40})$ definit astfel:

$$X_{40} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

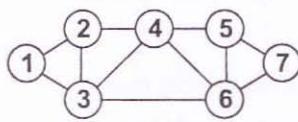
$$U_{40} = \{[1, 2], [1, 3], [1, 5], [2, 6], [2, 7], [3, 4], [3, 9], [4, 8], [8, 10]\}.$$

- Demonstrați că un arbore este un graf bipartit. Reciproca este adevărată?
- Scriți un program care citește dintr-un fișier text informații despre un graf neorientat (de pe primul rând, numărul de noduri ale grafului – n , iar de pe următoarele n rânduri matricea de adiacență a grafului) și care:
 - verifică dacă graful este un arbore;
 - dacă nu este arbore, verifică dacă prin eliminarea muchiilor poate fi făcut arbore; în caz afirmativ să se specifice căte muchii trebuie eliminate și care sunt aceste muchii.
 - dacă nu este arbore, verifică dacă prin adăugarea muchiilor poate fi făcut arbore; în caz afirmativ să se specifice căte muchii trebuie adăugate și care sunt aceste muchii.

2.8.2. Arborele parțial

2.8.2.1. Definiția arborelui parțial

Dacă un **graf parțial** al unui graf G este arbore, el se numește **arbore parțial** al grafului G .



Graful G_{41}

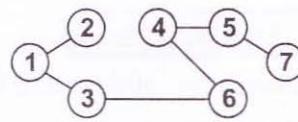


Fig. 48

Arbore parțial al grafului G_{41}

Teorema 22

Un graf G conține un **arbore parțial** dacă și numai dacă este un **graf conex**

Demonstrație. Notăm cele două propoziții, astfel:

- (1) – Graful G conține un arbore parțial.
- (2) – Graful G este conex.

Trebuie să demonstrează că (1) \Rightarrow (2) și (2) \Rightarrow (1).

(1) \Rightarrow (2). Să considerăm că graful G conține arborele parțial H . Din definiția arborelui rezultă că H este un graf conex. Din definiția arborelui parțial rezultă că H este graf parțial al grafului G . Deoarece graful G se obține prin adăugarea de muchii la un graf conex (H), este și el un graf conex.

(2) \Rightarrow (1). Dacă G este un graf conex minimal, înseamnă că este un arbore (din Teorema 21: (1) \Leftrightarrow (5)) și $H=G$. Dacă G nu este un graf conex minimal, înseamnă că există o muchie $[x,y]$ pe care o putem elimina astfel încât să obținem un graf parțial conex G_1 . Dacă graful parțial G_1 este un graf conex minimal, înseamnă că $H=G_1$; altfel se repetă procesul de eliminare a căte unei muchii până se obține un graf parțial conex minimal. Acesta va fi arborele parțial H .

2.8.2.2. Definiția arborelui parțial de cost minim

Considerăm un graf conex $G=(X,U)$, o funcție $c:U \rightarrow R_+$ care asociază fiecărei muchii u un număr real pozitiv $c(u)$, numit **costul** muchiei, și un graf $H=(X,V)$ care este graf parțial al grafului G ($V \subseteq U$). Funcția c se numește **funcția cost**. Definim:

Costul grafului este suma costurilor muchiilor grafului.

→ costul grafului G: $c(G) = \sum_u c(u)$

→ costul grafului parțial H: $c(H) = \sum_v c(v)$

Se numește **graf parțial de cost minim** al unui **graf G conex**, cu **funcția de cost c**, un **graf parțial conex H care are costul minim**.

Teorema 23

Graful parțial de cost minim al unui **graf conex G**, cu **funcția de cost c**, este un **arbore**.

Demonstrație – prin reducere la absurd. Dacă graful G este un arbore, atunci $H=G$, deoarece orice muchie am elimină din G și-ar pierde proprietatea de conexitate. Dacă graful G nu este un arbore, înseamnă că el conține un număr finit de grafuri parțiale conexe. Alegem dintre aceste grafuri parțiale graful H care este graful parțial de cost minim. Presupunem că acest graf parțial nu este arbore. Înseamnă că el conține cel puțin un ciclu și există o muchie $u=[x,y]$ pe care o putem suprima astfel încât să obținem un alt graf parțial conex H_1 . Rezultă că:

$$c(H)=c(H_1)-c(u)>c(H_1)$$

Inegalitatea $c(H)>c(H_1)$ contrazice faptul că graful parțial H este de cost minim. Înseamnă că graful parțial H este arbore.

Arboarele care este un **graf parțial de cost minim** al **grafului conex G**, cu **funcția de cost c**, se numește **arbore parțial de cost minim (APM)**.

Studiu de caz

Scop: identificarea aplicațiilor în care se folosește arboarele parțial de cost minim.

Enunțul problemei. O rețea de calculatoare este formată din 7 calculatoare server conectate între ele și calculatoare client conectate la servere. Unele servere pot fi legate direct, altele nu. Pentru ca rețeaua să funcționeze trebuie să se asigure transportul datelor între oricare două calculatoare server (două servere trebuie să comunice fie prin legătură directă, fie prin intermediul altor servere). Fiecare legătură directă dintre două servere are asociat un cost de construcție. Legăturile care se pot face între servere și costurile asociate acestor legături sunt prezentate în figura 49. Să se găsească o soluție de construire a rețelei de calculatoare (de a lega serverele între ele) astfel încât costul de realizare a rețelei să fie minim.

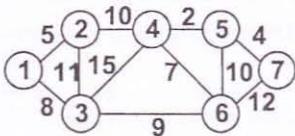


Fig. 49

G42 O rețea de servere este un graf neorientat G_{42} cu n noduri (în exemplu, $n=7$), în care nodurile sunt serverele, iar legăturile directe dintre două servere, muchiile grafului. Graful este conex (oricare ar fi două servere din rețea, între ele trebuie să existe un lanț care să le permită să comunice între ele). Găsirea soluției optime de conectare a serverelor, astfel încât costul financiar să fie minim, înseamnă determinarea unui graf conex de cost minim, adică găsirea arborului parțial de cost minim al grafului.

Observație. Orice rețea de transport (de calculatoare, rutieră, de cale ferată, aeriană, de telecomunicații, de canalizare etc.) este un graf neorientat conex. Construirea cu costuri financiare minime a unei astfel de rețele se rezolvă prin găsirea arborului parțial de cost minim al grafului asociat rețelei.



2.8.2.3. Algoritmi pentru determinare a arborelui parțial de cost minim

Algoritmii pentru determinarea arborelui parțial de cost minim folosesc **strategia greedy**:

PAS1. Se alege un subarbore al arborelui parțial de cost minim – H_{init} .

PAS2. Cât timp nu s-a format arborele parțial de cost minim **execută**:

PAS3. Se alege o **muchie sigură** din mulțimea muchiilor nealese (rămase).

PAS4. Se adaugă muchia la subarbore.

Muchia sigură trebuie să îndeplinească următoarele condiții:

- să aibă costul minim;
- să aparțină arborelui parțial de cost minim.

Determinarea arborelui parțial de cost minim se poate face prin:

- Algoritmul lui **Kruskal**.
- Algoritmul lui **Prim**.

Cei doi algoritmi diferă prin:

- modul în care se alege subarborele de la care se pornește;
- modul în care este găsită o muchie sigură.

a) Algoritmul lui Kruskal

Graful parțial al grafului $G=(X,U)$ care nu conține nici o muchie (are numai noduri izolate) este format din n arbori parțiali disjuncti:

$$\left. \begin{array}{l} H_1=(X_1,\emptyset) \text{ cu } X_1=\{x_1\} \\ H_2=(X_2,\emptyset) \text{ cu } X_2=\{x_2\} \\ \dots \\ H_k=(X_k,\emptyset) \text{ cu } X_k=\{x_k\} \\ \dots \\ H_n=(X_n,\emptyset) \text{ cu } X_n=\{x_n\} \end{array} \right\} \begin{array}{l} X_i \cap X_j = \emptyset, \forall i,j \text{ (} 1 \leq i \leq n, 1 \leq j \leq n \text{ și } i \neq j \text{)} \\ X_1 \cup X_2 \cup X_3 \cup \dots \cup X_n = X \end{array}$$

Se pornește de la doi arbori parțiali disjuncti H_i și H_j care se **unifică** prin adăugarea unei muchii sigure. **Muchia sigură** trebuie să îndeplinească următoarele condiții:

- să aibă costul minim;
- să nu formeze cicluri cu muchiile deja alese, adică extremitățile sale să aparțină celor două mulțimi disjuncte X_i și X_j – muchia este $[x_i, x_j]$ cu $x_i \in X_i$ și $x_j \in X_j$.

Structura de date folosită pentru implementarea grafului este **lista muchiilor**. Pentru a găsi mai ușor muchia cu costul minim, lista muchiilor este sortată crescător, după cost.

Algoritmul pentru determinarea APM este următorul:

PAS1. Pentru fiecare nod i din graf **execută**: se formează arborii parțiali H_i .

PAS2. Se sortează muchiile din U în ordinea crescătoare a costului c .

PAS3. Se alege muchia u cu costul minim.

PAS4. Se inițializează APM cu muchia u .

PAS5. Cât timp nu s-au selectat cele $n-1$ muchii **execută**:

PAS6. Se alege o muchie sigură din mulțimea muchiilor nealese (rămase) și se formează un arbore parțial cu această muchie.

PAS7. Se unifică APM cu arborele format cu muchia sigură.

Pentru a ține evidența arborilor parțiali H_i care se dezvoltă, se folosește lista L . În fiecare element $L(i)$ al listei se memorează numărul de ordine al arborelui parțial din care face parte nodul i . Inițial, existând n arbori parțiali H_i , fiecare dintre ei conținând un nod al grafului, elementele listei vor $L(i)=i$, pentru $i \in \{1, 2, 3, \dots, n\}$. Pe parcursul executării algoritmului, $L(i)=j$ înseamnă că nodul i aparține arborelui cu numărul de ordine j .

Pentru ca muchia $[x,y]$ care se alege să fie o muchie sigură trebuie să îndeplinească condițiile:
 → să aibă costul minim – muchiile se aleg în ordine, din lista muchiilor, unde ele sunt aranjate în ordinea crescătoare a costului;
 → extremitățile sale să aparțină la doi arbori parțiali diferenți – trebuie ca $L(x) \neq L(y)$.

După ce s-a găsit o muchie sigură, unificarea arborilor se face astfel:

- Dacă $L(x) < L(y)$, se adaugă arborele parțial din care face parte nodul y la arborele parțial din care face parte nodul x , adică se înlocuiesc toate elementele $L(i)=L(y)$ cu valoarea $L(x)$.
- Dacă $L(x) > L(y)$, se adaugă arborele parțial din care face parte nodul x la arborele parțial din care face parte nodul y , adică se înlocuiesc toate elementele $L(i)=L(x)$ cu valoarea $L(y)$.

Pentru graful G_{42} se obține arboarele parțial de cost minim din figura 50. Algoritmul se execută astfel:

Lista muchiilor

Muchia u	[1,2]	[1,3]	[2,3]	[2,4]	[3,4]	[3,6]	[4,5]	[4,6]	[5,6]	[5,7]	[6,7]
Costul c	5	8	11	10	15	9	2	7	10	4	12

Lista muchiilor sortată crescător după cost

Muchia u	[4,5]	[5,7]	[1,2]	[4,6]	[1,3]	[3,6]	[2,4]	[5,6]	[2,3]	[6,7]	[3,4]
Costul c	2	4	5	7	8	9	10	10	11	12	15

Lista arborilor parțiali – inițial este:

Nodul i	1	2	3	4	5	6	7
L[i]	1	2	3	4	5	6	7

Arborii parțiali	$H_i = (\{i\}, \emptyset)$ cu $1 \leq i \leq n$
------------------	---

Muchia cu costul minim este [4,5]

$L[4]=4$ $4 \neq 5 \Rightarrow L[4] \neq L[5]$

$L[5]=5$ $4 < 5 \Rightarrow L[5] = L[4]=4$

Nodul i	1	2	3	4	5	6	7
L[i]	1	2	3	4	5	6	7

$m=1$

$m \neq n-1=6$

Arborii parțiali sunt: $H_4=\{[4,5], \{[4,5]\}\}$ și $H_i=\{i, \emptyset\}$ cu $i \in \{1,2,3,6,7\}$

Următoarea muchie cu costul minim este [5,7]

$L[5]=4$ $4 \neq 7 \Rightarrow L[5] \neq L[7]$

$L[7]=7$ $4 < 7 \Rightarrow L[7] = L[5]=4$

Nodul i	1	2	3	4	5	6	7
L[i]	1	2	3	4	5	6	7

$m=2$

$m \neq n-1=6$

Arborii parțiali sunt: $H_4=\{[4,5,7], \{[4,5], [5,7]\}\}$ și $H_i=\{i, \emptyset\}$ cu $i \in \{1,2,3,6\}$

Următoarea muchie cu costul minim este [1,2]

$L[1]=1$ $1 \neq 2 \Rightarrow L[1] \neq L[2]$

$L[2]=2$ $1 < 2 \Rightarrow L[2] = L[1]=1$

Nodul i	1	2	3	4	5	6	7
L[i]	1	1	3	4	4	6	4

$m=3$

$m \neq n-1=6$

Arborii parțiali sunt: $H_1=\{[1,2], \{[1,2]\}\}$, $H_4=\{[4,5,7], \{[4,5], [5,7]\}\}$ și $H_i=\{i, \emptyset\}$ cu $i \in \{3,6\}$

Următoarea muchie cu costul minim este [4,6]

$L[4]=4$ $4 \neq 6 \Rightarrow L[4] \neq L[6]$

$L[6]=6$ $4 < 6 \Rightarrow L[6] = L[4]=4$

Nodul i	1	2	3	4	5	6	7
L[i]	1	1	3	4	4	4	4

$m=4$

$m \neq n-1=6$

Arborii parțiali sunt: $H_1=\{[1,2], \{[1,2]\}\}$, $H_4=\{[4,5,6,7], \{[4,5], [4,6], [5,7]\}\}$ și $H_3=\{\{3\}, \emptyset\}$

Următoarea muchie cu costul minim este [1,3]

$L[1]=1$ $1 \neq 3 \Rightarrow L[1] \neq L[3]$

$L[3]=3$ $1 < 3 \Rightarrow L[3] = L[1]=1$

Nodul i	1	2	3	4	5	6	7
L[i]	1	1	1	4	4	4	4

$m=5$

$m \neq n-1=6$

Arborii parțiali sunt: $H_1=\{\{1,2,3\}, \{[1,2], [1,3]\}\}$ și $H_4=\{[4,5,6,7], \{[4,5], [4,6], [5,7]\}\}$

Următoarea muchie cu costul minim este [3,6]

$L[3]=3$ $1 \neq 4 \Rightarrow L[3] \neq L[6]$

$L[6]=6$ $1 < 4 \Rightarrow L[6] = L[3]=1$

Nodul i	1	2	3	4	5	6	7
L[i]	1	1	1	1	1	1	1

$m=6$

$m=n-1=6$

și $L[7] = L[6] = L[5] = L[4] = L[3] = 1$

Arborele parțial de cost minim este: $H_1=\{\{1,2,3,4,5,6,7\}, \{[1,2], [1,3], [3,6], [4,5], [4,6], [5,7]\}\}$

Pentru implementarea algoritmului se folosesc următoarele variabile și structuri de date:

- variabilele n și m pentru numărul de noduri, respectiv numărul de muchii ale grafului;
- vectorul u , cu m elemente, pentru lista muchiilor; este un vector de structuri de tip `muchie`;
- vectorul L , cu n elemente, pentru lista arborilor parțiali;
- variabila ct pentru a calcula costul total al arborelui;
- variabila k pentru a număra muchiile adăugate la arborele parțial; inițial are valoarea 0 – arborele nu conține nici o muchie – valoarea sa finală este $n-1$ (numărul de muchii ale unui arbore cu n noduri);
- variabila i pentru indicele cu care se parcurge lista muchiilor;
- variabilele x și y pentru pentru a memora nodurile de la extremitățile unei muchii;

și subprogramele:

- funcția procedurală `citire` creează lista muchiilor u prin preluarea datelor din fișier – în fișierul `costapm.txt` pe primul rând se citește o valoare numerică ce reprezintă ordinul grafului, iar de pe următoarele rânduri trei valori numerice separate prin spațiu care reprezintă nodurile de la extremitățile unei muchii și costul asociat muchiei;
- funcția procedurală `init` inițializează lista L a arborilor parțiali H_i cu cei n arbori formați din cele n noduri izolate (arborii care nu conțin muchii);
- funcția procedurală `sortare` sortează lista muchiilor crescător după costul asociat;

```
#include <iostream.h>
fstream f("costapm.txt",ios::in);
int n,m,L[20];
struct muchie {int x,y,c;}; //x,y=noduri muchie; c=cost asociat muchiei
muchie u[20]; //u=listă muchiilor
void init() {for (int i=1;i<=n;i++) L[i]=i;}
void citire()
{int i=0,x,y,c; f>>n;
 while (f>>x>>y>>c) {i++; u[i].x=x; u[i].y=y; u[i].c=c;}
 f.close(); m=i;}
void sortare()
{int i,j; muchie aux;
 for (i=1;i<m;i++)
    for (j=i+1;j<=m;j++)
        if (u[i].c>u[j].c) {aux=u[i]; u[i]=u[j]; u[j]=aux;}}
void main()
{int i=1,j,k=0,x,y,ct=0; citire(); sortare(); init();
 cout<<"APM este format din muchiile: "<<endl;
 while (k<n-1) //cât timp nu s-au găsit cele n-1 muchii ale APM
 {if (L[u[i]].x!=L[u[i]].y)
    {k++; ct=ct+u[i].c; //se alege muchia i pentru a unifica doi arbori
     cout<<"["<<u[i].x<<","<<u[i].y<<"] ";
     x=L[u[i]].y; //se unifică cei doi arbori prin muchia [x,y]
     y=L[u[i]].x;
     for (j=1;j<=n;j++) if (L[j]==x) L[j]=y;}
    i++;} //se trece la muchia următoare din lista muchiilor
 cout<<endl<<"costul total= "<<ct;}
```

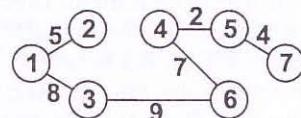


Fig. 50

Complexitatea algoritmului lui Kruskal

Pasul 1 are ordinul de complexitate $O(n)$. Pasul 2 are ordinul de complexitate în funcție de algoritmul de sortare ales. Algoritmii de sortare prin metoda selecției directe sau metoda

bulelor au ordinul de complexitate $O(m^2)$. Pasul 3 nu se poate preciza de câte ori se execută. În cazul cel mai defavorabil, se parcurg toate cele m muchii, deci se execută de m ori. În cazul pasului 3, pentru fiecare muchie aleasă, se parcurg toate cele n elemente ale listei L . Rezultă că pasul 3 are ordinul de complexitate $O(nxm)$. Ordinul de complexitate al algoritmului va fi: $O(n) + O(m^2) + O(nxm) = O(\max(m^2, nxm))$. Dacă graful are foarte multe muchii, atunci ordinul său de complexitate este $O(m^2)$.

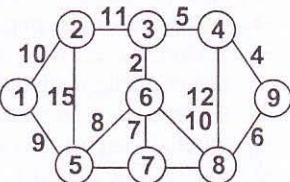


Fig. 51

G₄₃**Temă**

Pentru graful $G_{43}(X_{43}, U_{43})$ din figura 51, determinați APM executând algoritmul lui Kruskal. Verificați dacă soluția obținută este corectă executând programul pentru graful G_{43} .

b) Algoritmul lui Prim

Algoritmul lui Prim se aseamănă cu algoritmul lui Dijkstra pentru determinarea drumurilor de lungime minimă într-un graf. În graful $G=(X,U)$, subarborele de la care se pornește H_{init} este format dintr-un nod inițial numit rădăcină (r – care se comunică algoritmului împreună cu datele despre graf) și crește până acoperă toate nodurile din mulțimea X . Dacă arborele parțial care se dezvoltă este $H=(A,V)$, definim graful $G_A=(X-A, \emptyset)$. Nodurile care fac parte din G_A sunt memorate într-o **coadă de priorități** Q . Coada de priorități se deosebește de o coadă simplă, prin aceea că din ea nu se extrage primul element introdus, ci elementul cu valoarea cea mai mică (sau cea mai mare, în funcție de prioritatea aleasă).

În coada de priorități Q se memorează pentru fiecare nod i , eticheta nodului $j \in A$ cu care formează muchia cu costul minim. La fiecare adăugare a unui nou nod la arbore, acesta trebuie eliminat din coada de priorități, iar coada trebuie reactualizată pentru toate nodurile rămase, deoarece s-a modificat mulțimea A , iar un nod din coada de așteptare poate să aibă muchia cu costul minim cu noul nod adăugat la arborele parțial. Arborele care se dezvoltă este memorat în lista H , astfel: pentru fiecare nod i care se adaugă la arborele parțial, se memorează nodul $j \in A$ cu care este legat prin muchie. Inițial $H_{init}=\{r\}, \emptyset$, $G_A=(X-\{r\}, \emptyset)$, în lista H toate elementele au valoarea 0, iar în coada de priorități Q , fiecarui nod $i \neq r$ i se va atribui valoarea r ($Q(i)=r$).

Muchia sigură trebuie să îndeplinească următoarele condiții:

- să aibă costul minim;
- să nu formeze cicluri cu muchiile deja alese, adică să unească un nod din H cu un nod izolat din G_A – extremitățile sale aparțin celor două mulțimi disjuncte A și $X-A$ (muchia este $[x_i, x_j]$ cu $x_i \in A$ și $x_j \in X-A$).

Structura de date folosită pentru implementarea grafului este **matricea costurilor**.

Algoritmul pentru determinarea APM este următorul:

- PAS1.** Se inițializează APM cu nodul rădăcină r .
- PAS2.** Se inițializează lista H , astfel: **pentru** fiecare nod i din graf **execută**: $H(i) \leftarrow 0$.
- PAS3.** Se inițializează coada de priorități Q , astfel: $Q(r) \leftarrow 0$ și **pentru** fiecare nod $i \neq r$ din graf **execută**: $Q(i) \leftarrow r$.
- PAS4.** **Cât timp** nu s-au selectat cele $n-1$ muchii **execută**:

 - PAS5.** Se alege o muchie sigură din mulțimea muchiilor nealese – muchia $[i,j]$ cu $i \in A$ și $j \in X-A$
 - PAS6.** Se adaugă muchia la lista H : $H(j) \leftarrow Q(j)$.

- PAS7.** Se elimină din coada de priorități nodul j adăugat la arbore, atribuindu-i valoarea 0: $Q(j) \leftarrow 0$.
- PAS8.** Se actualizează coada de priorități pentru nodurile nealese, astfel pentru fiecare nod i din Q execută: cauță nodul $j \in A$ cu care formează muchia cu costul minim.

Coada de priorități Q este implementată cu un vector cu n elemente. Inițial:

$$Q(i) = \begin{cases} 0 & \text{dacă } i = r \\ r & \text{dacă } i \neq r \end{cases}$$

În timpul execuției algoritmului:

$$Q(i) = \begin{cases} 0 & \text{dacă } i \in A \\ r & \text{dacă } i \notin A \text{ și nu } \exists j \in A \text{ a.t. } [i, j] \in U \\ j & \text{dacă } i \notin A \text{ și } \exists j \in A \text{ a.t. } [i, j] \in U \text{ și } \text{cost}([i, j]) = \min \end{cases}$$

La terminarea execuției algoritmului: $Q(i)=0$, pentru orice i ($1 \leq i \leq n$).

Lista H este implementată cu un vector cu n elemente. Inițial: $H(i)=0$, pentru orice i ($1 \leq i \leq n$).

În timpul execuției algoritmului:

$$H(i) = \begin{cases} 0 & \text{dacă } i = r \text{ sau } i \notin A \\ j & \text{dacă } i \in A, \text{ iar } [i, j] \in H \end{cases}$$

La terminarea execuției algoritmului:

$$H(i) = \begin{cases} 0 & \text{dacă } i = r \\ j & \text{dacă } i \neq r \text{ și } [i, j] \in H \end{cases}$$

Pentru ca muchia $[i, j]$ care se alege să fie o muchie sigură trebuie să îndeplinească condițile:

→ să aibă costul minim – se cauță nodul j pentru care în matricea costurilor elementul $a[Q(i)][j]$ are valoarea minimă;

→ extremitățile sale să aparțină la doi arbori parțiali diferiți – trebuie ca $Q(i) \neq 0$ și $Q(j) = 0$.

După ce s-a găsit o muchie sigură, ea este adăugată la arbore prin $T(j)=Q(j)$.

Pentru graful G_{42} din figura 49, considerând $r=1$, algoritmul se execuțiază astfel:

Lista muchiilor

Muchia u	[1,2]	[1,3]	[2,3]	[2,4]	[3,4]	[3,6]	[4,5]	[4,6]	[5,6]	[5,7]	[6,7]
Costul c	5	8	11	10	15	9	2	7	10	4	12

Arboarele parțial (H) și coada de priorități Q – inițial sunt:

Nodul i	1	2	3	4	5	6	7
$H[i]$	0	0	0	0	0	0	0
$Q[i]$	0	1	1	1	1	1	1
Costul muchiei $[i, Q[i]]$	-	5	8	VMAX	VMAX	VMAX	VMAX

Arboarele inițial
 $H_{init} = (\{1\}, \emptyset)$
 $\min=5$
 $j=2$

Muchia cu costul minim este $[Q[j], j] = [Q[2], 2] = [1, 2]$

Nodul i	1	2	3	4	5	6	7
$H[2]=Q[2]$	0	1	0	0	0	0	0
$Q[2]=0$	0	0	1	1	1	1	1
Costul $[i, Q[i]]$	-	-	8	VMAX	VMAX	VMAX	VMAX
Actualizare	Q	Costul $[i, Q[i]]$	Q[i]	0	0	1	2
Q	Costul $[i, Q[i]]$	-	-	8	10	VMAX	VMAX

$\min=8$
 $j=3$

Arboarele parțial $H=\{1, 2, \{1, 2\}\}$ și $X-A=\{3, 4, 5, 6, 7\} \rightarrow m=1$

Muchia cu costul minim este $[Q[j], j] = [Q[3], 3] = [1, 3]$

		Nodul i	1	2	3	4	5	6	7
H[3]=Q[3]		H[i]	0	1	1	0	0	0	0
Q[3]=0		Q[i]	0	0	0	1	1	1	1
Actualizare		Costul $[i, Q[i]]$	-	-	-	10	VMAX	VMAX	VMAX
Q		Q[i]	0	0	0	2	1	3	1
Costul $[i, Q[i]]$		-	-	-	10	VMAX	9	VMAX	

min=9
j=6

Arborele parțial $H = \{1, 2, 3, \{1, 2\}, \{1, 3\}\}$ și $X-A = \{4, 5, 6, 7\} \rightarrow m=2$

Muchia cu costul minim este $[Q[j], j] = [Q[6], 6] = [3, 6]$

		Nodul i	1	2	3	4	5	6	7
H[6]=Q[6]		H[i]	0	1	1	0	0	3	0
Q[6]=0		Q[i]	0	0	0	2	1	0	1
Actualizare		Costul $[i, Q[i]]$	-	-	-	10	VMAX	-	VMAX
Q		Q[i]	0	0	0	6	6	0	6
Costul $[i, Q[i]]$		-	-	-	7	10	-	12	

min=7
j=4

Arborele parțial $H = \{1, 2, 3, 6, \{1, 2\}, \{1, 3\}, \{3, 6\}\}$ și $X-A = \{4, 5, 7\} \rightarrow m=3$

Muchia cu costul minim este $[Q[j], j] = [Q[4], 4] = [6, 4]$

		Nodul i	1	2	3	4	5	6	7
H[4]=Q[4]		H[i]	0	1	1	6	0	3	0
Q[4]=0		Q[i]	0	0	0	0	6	0	6
Actualizare		Costul $[i, Q[i]]$	-	-	-	-	10	-	12
Q		Q[i]	0	0	0	0	4	0	6
Costul $[i, Q[i]]$		-	-	-	-	2	-	12	

min=2
j=5

Arborele parțial $H = \{1, 2, 3, 4, 6, \{1, 2\}, \{1, 3\}, \{3, 6\}, \{4, 6\}\}$ și $X-A = \{5, 7\} \rightarrow m=4$

Muchia cu costul minim este $[Q[j], j] = [Q[5], 5] = [4, 5]$

		Nodul i	1	2	3	4	5	6	7
H[5]=Q[5]		H[i]	0	1	1	6	4	3	0
Q[5]=0		Q[i]	0	0	0	0	0	0	6
Actualizare		Costul $[i, Q[i]]$	-	-	-	-	-	-	12
Q		Q[i]	0	0	0	0	0	0	5
Costul $[i, Q[i]]$		-	-	-	-	-	-	4	

min=4
j=7

Arborele parțial $H = \{1, 2, 3, 4, 5, 6, \{1, 2\}, \{1, 3\}, \{3, 6\}, \{4, 5\}, \{4, 6\}\}$ și $X-A = \{7\} \rightarrow m=5$

Muchia cu costul minim este $[Q[j], j] = [Q[7], 7] = [5, 7]$

		Nodul i	1	2	3	4	5	6	7
H[7]=Q[7]		H[i]	0	1	1	6	4	3	5
Q[7]=0		Q[i]	0	0	0	0	0	0	0
Actualizare		Costul $[i, Q[i]]$	-	-	-	-	-	-	-
Q		Q[i]	0	0	0	0	0	0	0
Costul $[i, Q[i]]$		-	-	-	-	-	-	-	

X-A=Ø

Arborele parțial $H = \{1, 2, 3, 4, 5, 6, 7, \{1, 2\}, \{1, 3\}, \{3, 6\}, \{4, 5\}, \{4, 6\}, \{5, 7\}\}$ → $m=6$

Pentru implementarea algoritmului se folosesc următoarele variabile și structuri de date:

- variabilele n și x pentru numărul de noduri, respectiv pentru nodul rădăcină;
- matricea pătratică A , cu dimensiunea n , pentru matricea costurilor asociată grafului;
- vectorul Q , cu n elemente, pentru coada de priorități;
- vectorul H , cu n elemente, pentru lista muchiilor din APM;
- variabila c pentru a calcula costul total al arborelui;

- variabila k pentru a număra muchiile adăugate la arborele parțial; inițial are valoarea 0 – arborele nu conține nici o muchie – valoarea sa finală este $n-1$ (numărul de muchii ale unui arbore cu n noduri);
- variabilele i și j pentru indicele cu care se parcurge coada de priorități, respectiv pentru nodul care se adaugă la arbore;

și subprogramele:

- funcția procedurală `init_mc` inițializează matricea costurilor;
- funcția procedurală `citire_mc` actualizează matricea costurilor cu datele din fișier;
- funcția procedurală `init_Q` inițializează coada de priorități Q ;
- funcția operand `muchie` cauță o muchie sigură în multimea muchiilor nealese;
- funcția procedurală `actualizeaza_Q` actualizează coada de priorități Q după ce a fost adăugată muchia la arborele parțial;
- funcția procedurală `afisare` afișează APM.

```
#include <iostream.h>
fstream f("costapm.txt",ios::in);
int a[50][50],Q[50],H[50],n,r;
const int VMAX=5000;
void init_mc()
{int i,j; f>>n;
 for(i=1;i<=n;i++)
  for(j=1;j<=n;j++) if (i!=j) a[i][j]=VMAX; }
void citire_mc()
{int i,j,c; while (f>>i>>j>>c) a[i][j]=c; a[j][i]=c; f.close();}
void init_Q() {for (int i=1;i<=n;i++) if (i!=r) Q[i]=r;}
int muchie()
{int i,j,min=VMAX;
 for(i=1;i<=n;i++)
  if (Q[i]!=0 && a[Q[i]][i]<min) {min=a[Q[i]][i]; j=i;}
 return j;}
void actualizeaza_Q(int j)
{for(int i=1;i<=n;i++) if (Q[i]!=0 && a[i][Q[i]]>a[i][j]) Q[i]=j;}
void afisare()
{cout<<"APM este format din muchiile: "<<endl;
 for(int i=1;i<=n;i++) if (H[i]!=0) cout<<"["<<H[i]<<","<<i<<"]" <<endl;
 void main()
{int i,j,k=0,ct=0; init_mc(); citire_mc();
 cout<<"Nodul initial: "; cin>>r; init_Q();
 while (k<n-1) //cât timp nu s-au găsit cele n-1 muchii ale APM
 {j=muchie(); //alege o muchie sigură
  H[j]=Q[j]; ct+=a[Q[j]][j]; //adăugă la arbore nodul și muchia
  Q[j]=0; //elimină din coada de priorități nodul adăugat
  actualizeaza_Q(j); //actualizează coada de priorități
  k++;}
 cout<<"costul total= "<<ct<<endl; afisare();}
```

Complexitatea algoritmului lui Prim

Pasul 2 și pasul 3 au ordinul de complexitate $O(n)$. Pasul 4 se execută de $n-1$ ori. În cazul pasului 3, pentru fiecare muchie aleasă, se execută pasul 5 și pasul 8 de n ori. Rezultă că Pasul 3 are ordinul de complexitate $O((n-1) \times (n+n)) = O(n^2)$. Ordinul de complexitate al algoritmului va fi: $O(n) + O(n) + O(n^2) = O(n^2)$.

Tema

Pentru graful G_{43} din figura 51, determinați APM executând algoritmul lui Prim. Verificați dacă soluția obținută este corectă executând programul pentru graful G_{43} .

2.8.2.4. Aplicații practice

1. Trebuie să se construiască o rețea de autostrăzi care să lege cele mai importante orașe din țară. Fiecare tronson de autostradă care leagă două orașe are un cost de construire. Să se determine rețeaua de autostrăzi astfel încât toate orașele să fie legate prin autostrăzi și costul construirii ei să fie minim.
2. Într-o localitate trebuie construită o rețea de canalizare care să lege n locuințe și să comunice cu două puncte de deversare. Fiecare tronson de rețea are un cost de construire. Să se determine rețeaua de canalizare astfel încât toate locuințele să aibă acces la ea și costul construirii ei să fie minim.

2.8.3. Arborele cu rădăcină

2.8.3.1. Definiția arborelui cu rădăcină

Se numește **arbore cu rădăcină** un arbore A în care există un nod privilegiat numit **nod rădăcină**.

A1 **Terminologie** – Multii dintre termenii folosiți sunt preluati din terminologia arborilor genealogici sau a arborilor din natură (ca exemplu, arborele A₁ figura 52).

- Muchiile unui arbore se mai numesc **ramuri** sau **arce**.
- **Nodul rădăcină** mai este numit **vârf** sau **tulpină**. În nodul rădăcină nu intră nici un arc.
- Într-un nod intră un singur arc (exceptând rădăcina) care îl leagă de un alt nod numit **părinte** sau **predecesor**.
- Dintr-un nod pot să iasă niciunul, unul sau mai multe arce care îl leagă de un alt nod numit **fiu** sau **succesor**.
- Nodurile fără succesi (din care nu ieșe nici un arc) se numesc **frunze** sau **noduri terminale**. Nodurile care nu sunt terminale se mai numesc **noduri de ramificare**.
- Două noduri adiacente din arbore sunt în relația **tată-fiu** sau **părinte-fiu**, iar muchiile sunt **legături de tip tată-fiu**. Între mai multe noduri se poate stabili o relație de tipul **fiul fiului ... fiului sau tatăl tatălui... tatălui**. În primul caz se poate spune că nodul este **descendent** sau **urmaș** al unui alt nod, iar în al doilea caz că nodul este **ascendent** sau **strămoș** al unui nod.
- Două noduri care descind direct din același nod tată se numesc **noduri frate**.
- **Ordinul unui nod** este dat de numărul de descendenți direcți.
- Nodurile sunt organizate pe **niveluri**. Numerotând nivelurile nodurilor, rădăcina se găsește pe nivelul 0, descendenții ei pe nivelul 1, descendenții acestora pe nivelul 2 etc. **Nivelul unui nod** este egal cu numărul de noduri parcurse pe calea de la rădăcina la el.
- Un arbore cu rădăcină este **arbore vid** (arbore nul) dacă nu are nici un nod.

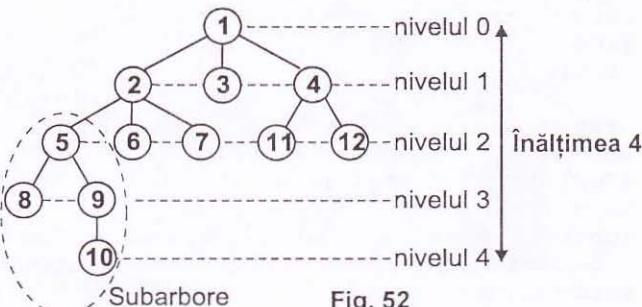


Fig. 52

→ Înălțimea unui arbore este data de maximul dintre nivelurile nodurilor terminale (lungimea celui mai lung lanț care pornește din rădăcină).



Tema

- Pentru arborele cu rădăcină A_1 din figura 52 precizați:
 - eticheta nodului rădăcină;
 - numărul de frunze și etichetele lor;
 - etichetele nodurilor care se găsesc pe nivelul 2;
 - etichetele fiilor nodului 2;
 - etichetele fratilor nodului 5;
 - eticheta părintelui nodului 9;
 - numărul de frați ai nodului 7.

2. Un arbore cu rădăcină A_2 (X, U) este definit astfel:

$$X = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

$$U = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 6\}, \{3, 5\}, \{3, 7\}, \{3, 9\}, \{4, 8\}, \{8, 10\}\}.$$



A₂

Precizați:

- dacă eticheta nodului rădăcină este 1, câte frunze are arborele și care sunt aceste frunze;
- dacă eticheta nodului rădăcină este 2, câți frați are nodul cu eticheta 1 și pe ce nivel se găsește nodul cu eticheta 10;
- dacă eticheta nodului rădăcină este 10, ce înălțime are arborele, cine este părințele nodului cu eticheta 1 și care sunt fiii nodului cu eticheta 3.

Caracteristici:

- Nodul rădăcină este un nod considerat privilegiat. El nu are părinte (ascendent), ci numai fi (descendenți). Este nodul de la care se consideră că pornesc ramurile către rădăcinile altor arbori. **Orice nod al arborelui poate fi considerat nod rădăcină.**
- Un arbore A este fie vid, fie format dintr-un nod rădăcină R căruia îi este atașat un număr finit de arbori. Acești arbori sunt subordonați rădăcinii și se numesc **subarbori** ai arborelui A. **Orice nod dintr-un arbore este rădăcina unui subarbore.**
- Între doi subarbori nu pot exista decât două tipuri de relații:
 - relația de incluziune** – unul dintre subarbori este subarborele celuilalt;
 - relația de excluziune** – cei doi subarbori nu au noduri comune, dar aparțin aceluiași arbore.
- Accesul de la rădăcina unui arbore (subarbore) nevid la oricare nod, înseamnă parcurserea unui lanț format din **m** arce, care trec prin **n** noduri ($m=n+1$), **n** reprezentând nivelul pe care se găsește nodul față de rădăcină.

Definiții recursive:

- Definiția arborelui.** Un arbore nevid este o mulțime finită A de noduri care au următoarele proprietăți:
 - Există un nod care poate fi considerat **nod rădăcină**.
 - Celealte noduri pot fi repartizate în i ($i \geq 0$) mulțimi disjuncte (A_1, A_2, \dots, A_i), fiecare dintre aceste mulțimi fiind considerată la rândul lor arbore.
- Definiția înălțimii.** Înălțimea unui arbore este egală cu **1+maximul dintre înălțimile subarborilor săi.**



Tema

- Dacă nodul cu eticheta 2 are patru frați, iar nodul cu eticheta 1 este părintele lui, ce ordin are nodul cu eticheta 1?
- Pentru arborele cu rădăcină A_1 din figura 52 precizați:
 - cât subarbori are nodul rădăcină – pentru fiecare subarbore precizați rădăcina;

- câți subarbore are nodul cu eticheta 2 – pentru fiecare subarbore precizați rădăcina;
 - câți subarbore are nodul cu eticheta 10 – precizați cine este acest subarbore;
 - în ce relație sunt subarborele cu rădăcina în nodul cu eticheta 2 și subarborele cu rădăcina în nodul cu eticheta 4;
 - în ce relație sunt subarborele cu rădăcina în nodul cu eticheta 2 și subarborele cu rădăcina în nodul cu eticheta 9.
3. Pentru arborele cu rădăcină A₂ definit anterior precizați:
- dacă eticheta nodului rădăcină este 1, în ce relație sunt subarborele cu rădăcina în nodul cu eticheta 2 și subarborele cu rădăcina în nodul cu eticheta 8;
 - dacă eticheta nodului rădăcină este 3, câți subarbore are nodul rădăcină – pentru fiecare subarbore precizați rădăcina;
 - dacă eticheta nodului rădăcină este 2, câți subarbore are nodul cu eticheta 9 – precizați cine este acest subarbore;
 - dacă eticheta nodului rădăcină este 3, câți subarbore are nodul cu eticheta 2 – pentru fiecare subarbore precizați rădăcina;
 - dacă eticheta nodului rădăcină este 4, în ce relație sunt subarborele cu rădăcina în nodul cu eticheta 1 și subarborele cu rădăcina în nodul cu eticheta 3.

Se numește arbore ordonat un arbore cu rădăcină în care fiile fiecarui nod sunt ordonați.

Observație. Într-un arbore ordonat, dacă un nod are k fi, atunci există un prim fiu, un al doilea fiu, ..., un al k-lea fiu.

Se numește arborescență sau structură arborescentă un arbore cu rădăcină în care s-a stabilit nodul rădăcină.

Pentru a înțelege deosebirea dintre tipurile de arbori enumerate, vom considera pentru exemplificare un arbore cu 3 noduri etichetate cu A, B și C și vom pune în evidență numărul de arbori diferenți care se pot forma cu aceste noduri.

→ **Arbore liber.** Se pot obține 3 arbori liberi diferenți. În figura 53 sunt prezentate acești arbori.

→ **Arbore cu rădăcină.** Se pot obține 9 arbori cu rădăcină diferenți. Există trei moduri de a alege rădăcina (nodul A, nodul B sau nodul C). Pentru un nod rădăcină ales, se pot obține trei arbori diferenți. De exemplu, dacă se consideră nodul A ca nod rădăcină, există trei moduri diferenți de a repartiza în arbore nodurile B și C: nodul A are doi fi (B și C) sau un singur fiu: B sau C, obținându-se trei arbori diferenți (cei prezenți în figura 54). În total se obțin $3 \times 3 = 9$ arbori cu rădăcină diferenți.

→ **Arbore cu rădăcină ordonați.** Se pot obține 12 arbori cu rădăcină ordonați diferenți. Ca și în cazul precedent, există trei moduri de a alege rădăcina. Pentru un nod rădăcină ales, se pot obține patru arbori diferenți. De exemplu, dacă se consideră nodul A ca nod rădăcină, există patru moduri diferenți de a repartiza în arbore nodurile B și C:

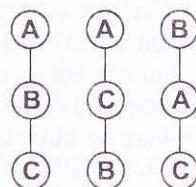


Fig. 53

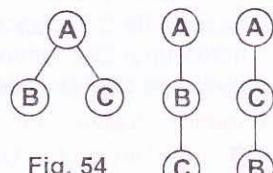


Fig. 54

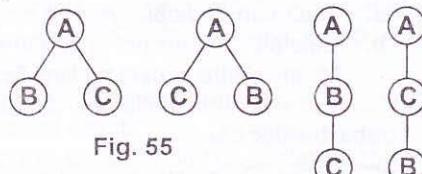


Fig. 55

nodul A are doi fi (B și C sau C și B – în acest caz ordinea nodurilor B și C contează), sau un singur fiu: B sau C, obținându-se patru arbori diferenți (cei prezenți în figura 55). În total se obțin $3 \times 4 = 12$ arbori cu rădăcină ordonați diferenți.

→ **Structuri arborescente.** Pentru un nod rădăcină ales (de exemplu nodul B) se pot obține 3 structuri arborescente diferențe deoarece există trei moduri diferențe de a repartiza în arbore nodurile A și C: nodul B are doi fi (A și C) sau un singur fiu: A sau C (figura 56).

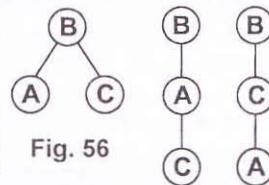


Fig. 56

Se numește arbore pozitional un arbore cu rădăcină în care este precizată poziția fiecărui fiu.

Observație. Într-un arbore pozitional, fiii fiecărui nod sunt etichetați cu numere întregi pozitive consecutive, iar dacă lipsește fiul cu eticheta k, această etichetă nu se atribuie nici unui fiu (arborele A₃ din figura 57).

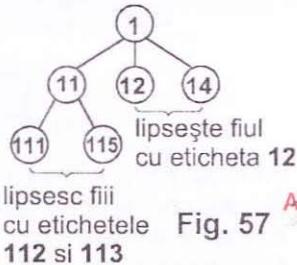


Fig. 57

Studiu de caz

Scop: identificarea colecțiilor de date care pot fi reprezentate prin arbori cu rădăcină.

Enunțul problemei 1. O firmă are mai mulți angajați. Între angajați există fie relații de subordine, fie relații de colaborare. Directorul firmei dă o dispoziție pe care o transmite subordonatilor săi direcți. Aceștia primesc simultan dispoziția și o transmit mai departe subordonatilor direcți și.a.m.d. Transmiterea dispoziției de la un nivel ierarhic la altul necesită același timp t. Să se determine în cât timp ajung să cunoască dispoziția dată toți angajații firmei (figura 58).

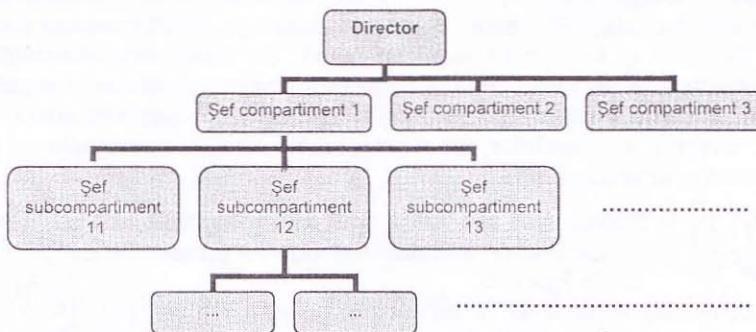


Fig. 58

Firma are o structură ierarhică, iar relațiile dintre angajați pot fi reprezentate cu ajutorul unei **arborescente**, în care angajații sunt nodurile, iar arcele relațiile de subordonare. Rădăcina arborelui este directorul. Între un angajat care are în subordine un alt angajat există o legătură de tip tată-fiu. Între angajații subordonati aceluiași angajat există relații de colaborare. Acești angajați sunt noduri frate. Timpul T necesar ca dispoziția să ajungă de la director la toți angajații este dat de produsul dintre timpul t necesar pentru transmiterea de la un nivel la altul și înălțimea h a arborelui: $T=t \times h$.

Enunțul problemei 2. Un produs este realizat din mai multe componente. Fiecare componentă la rândul său este realizată prin asamblarea altor componente. Atât produsul cât și componentele sunt caracterizate printr-o listă de atribute (proprietăți): denumire,

cod, preț, funcția realizată, costul asamblării, firma care o produce etc. Să se determine prețul produsului (figura 59).

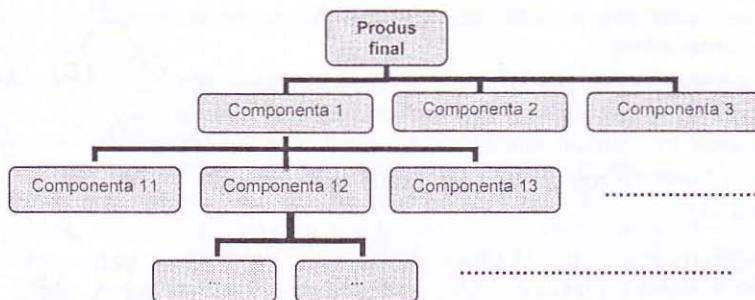


Fig. 59

Produsul are o structură jerarhică, iar relațiile dintre componente pot fi reprezentate cu ajutorul unei **arborescențe**, în care componentele sunt nodurile, iar arcele relațiile de apartenență a unei componente la o altă componentă. Rădăcina arborelui este produsul final. Între o componentă și componenta pe care o are în subordine există o legătură de tip tată-fiu. Componentele care fac parte din aceeași componentă sunt frați. Pentru stabilirea costului total al produsului trebuie parcurs arborele, ca să se calculeze costul fiecărei componente. Costul **c** al unei componente se obține prin adunarea prețului de achiziție **p** la costul asamblării **a**: $c = p + a$. Parcurgerea arborelui se face pornind de la ultimul nivel către rădăcină. Rădăcina se prelucrează ultima, după ce au fost prelucrați toți subarborii subordonata ei.

Observație. Pentru studierea proprietăților unui obiect, orice obiect poate fi descompus la rândul său în obiecte mai simple. Fiecare obiect, la rândul său, poate fi caracterizat printr-o listă de proprietăți. Procesul de descompunere poate să continue pe mai multe niveluri. El este finit și se termină după un număr de etape care este dependent de problema care trebuie rezolvată. Procesul de descompunere este un **proces recursiv** (un obiect este compus din alte obiecte). Prin procesul de descompunere se stabilește o **ierarhie a obiectelor**, iar reprezentarea acestei descompuneri se poate face printr-o **arborescență**.



1. Reprezentați sub formă unui arbore cu rădăcină, relația de inclusiune a mulțimilor prezentate în figura 60.
2. Doriți să cumpărați un calculator și dispuneți de un anumit buget. Pentru a avea un calculator mai performant, alegeti varianta de a cumpăra componentele și de a le asambla singur. Calculatorul este un produs format din mai multe componente (unitatea centrală, monitorul, tastatura, mouse-ul etc.) care la rândul lor pot fi descompuse în alte componente (de exemplu, unitatea centrală este formată din carcăsă, placă de bază, unitatea de hard-disc, unitatea de compact disc sau de DVD, etc.). Desenați arborele structurii de componente a calculatorului.
3. Desenați un arbore genealogic al familiei care să prezinte **filiatia** (descendenții direcți ai unei persoane). Se va lua ca nod rădăcină unul dintre stră-străbunici. Identificați pe acest arbore relații de rudenie de tip: fiu, nepot, frate și văr.

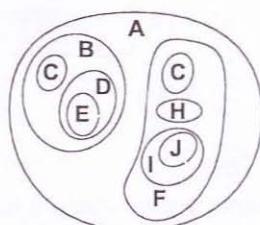


Fig. 60

4. Cuprinsul unei cărți are o structură arborescentă. Desenați structura arborescentă a capitolului „Implementarea structurilor de date” din acest manual.
5. Structura dosarelor cu fișiere de pe hard-disc, gestionate de sistemul de operare Windows, este o structură arborescentă. Desenați arborele dosarelor de pe hard-disc. Fișierele vor fi nodurile terminale.
6. O matrice cu n linii și m coloane este o structură de date ierarhizată care poate fi reprezentată pe trei niveluri, ca o arborescentă, astfel: nodul rădăcină este matricea, fiile nodului matrice sunt liniile, iar fiile fiecărei linii, elementele de pe linia respectivă. Desenați arborele cu rădăcină al unei matrice cu patru linii și trei coloane. Precizați ce tip de arbore cu rădăcină este.

2.8.3.2. Implementarea arborelui cu rădăcină

Implementarea structurilor de date de tip arbore cu rădăcină se poate face prin:

- matrice de adiacență;
- listă de adiacență;
- referințe descendente – legătura de **tip tată**;
- referințe ascendente – legătura de **tip părinte nod terminal**.

Observație. Arborele, fiind un graf neorientat cu anumite proprietăți, pentru implementarea sa statică se pot folosi aceleași metode ca și la grafuri, dar aceste implementări sunt inefficiente. Arborilor le sunt specifice implementarea cu **legătura de tip tată** și implementarea cu **legătura de tip părinte nod terminal**. În următoarele implementări se consideră că arborele are n noduri și rădăcina are eticheta r .

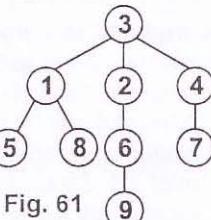


Fig. 61

a) Implementarea prin referințe descendente

Legătura de tip tată. Arborele este reprezentat sub formă unui vector t cu n componente în care se memorează, pentru fiecare nod, eticheta părintelui său. Algoritmul de construire a vectorului este următorul:

PAS1. Pentru fiecare nod i din arbore **execută**:

PAS2. Dacă nodul $i=r$, atunci $t[i] \leftarrow 0$; altfel, $t[i] \leftarrow j$, unde j reprezintă nodul părinte al nodului i .

Exemplu. Vectorul **tată** pentru arborele A_4 din figura 61 este:

Nod (indice i)	1	2	3	4	5	6	7	8	9
Părinte ($t[i]$)	3	3	0	3	1	2	4	1	6

Observație. Din vectorul **tată** puteți obține următoarele informații

- eticheta nodului rădăcină – indicele i pentru care $t[i]=0$;
- etichetele nodurilor terminale – nodurile j a căror etichetă nu există în vectorul t ;
- eticheta părintelui unui nod $j - t[j]$;
- etichetele filor unui nod j – indicii i pentru care $t[i]=j$;
- etichetele fraților unui nod j – indicii i pentru care $t[i]=t[j]$.

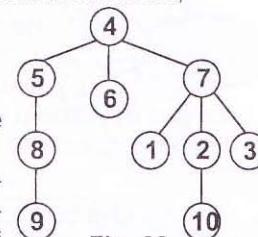


Fig. 62

Temă



1. Reprezentați sub formă vectorului **tată** arborele cu rădăcină A_5 din figura 62.
2. Scrieți un program care să citească dintr-un fișier informații despre matricea de adiacență a unui arbore cu rădăcină (de pe primul rând – numărul de noduri n și eticheta rădăcinii r , iar de pe următoarele n rânduri – matricea de adiacență) și care să scrie vectorul **tată** al arborelui într-un alt fișier.

A4

A5

3. Scrieți un program care să citească din fișierul creat anterior vectorul tată al arborelui cu rădăcină și să afișeze următoarele informații:
- eticheta nodului rădăcină;
 - numărul de noduri terminale și etichetele lor;
 - pentru fiecare nod, eticheta părintelui, numărul de fii și etichetele lor, și numărul de frați și etichetele lor.

b) Implementarea prin referințe ascendentă

Legătura de tip părinte nod terminal. Arborele este reprezentat sub forma a doi vectori cu $n-1$ componente: vectorul t în care sunt memorate nodurile, în ordine, pornind de la nodurile terminale, și vectorul pt în care sunt memorate nodurile părinte ale nodurilor terminale. Algoritmul de construire a celor doi vectori este următorul:

PAS1. Pentru fiecare indice i de la 1 la $n-1$ execută:

PAS2. Se caută nodul terminal cu eticheta cea mai mică.

PAS3. Se atribuie această etichetă lui $t[i]$.

PAS4. Se atribuie lui $pt[i]$ eticheta nodului părinte al nodului terminal $t[i]$.

PAS5. Se elimină din arbore nodul terminal $t[i]$.

Exemplu. Pentru arborele A_4 din figura 61, cei doi vectori se completează astfel (figura 63):

- Inițial nodurile terminale au etichetele: 5, 8, 9 și 7. Se alege nodul terminal cu eticheta cea mai mică (5). Se scrie eticheta sa în primul element al vectorului t , iar eticheta părintelui său (1) în primul element al vectorului pt . Se înlătură nodul 5 din arbore.
- Nodurile terminale au etichetele: 8, 9 și 7. Se alege nodul terminal cu eticheta cea mai mică (7). Se scrie eticheta sa în al doilea element al vectorului t , iar eticheta părintelui său (4) în al doilea element al vectorului pt . Se înlătură nodul 7 din arbore.
- Nodurile terminale au etichetele: 8, 9 și 4. Se alege nodul terminal cu eticheta cea mai mică și a părintelui său în vectorul pt , urmată de eliminarea din arbore a nodului adăugat, continuă până când în arbore nu mai rămâne decât nodul care are cea mai mare etichetă.
- Procedeul de adăugare în vectorul t a nodului cu eticheta cea mai mică și a părintelui său în vectorul pt , urmată de eliminarea din arbore a nodului adăugat, continuă până când în arbore nu mai rămâne decât nodul care are cea mai mare etichetă.

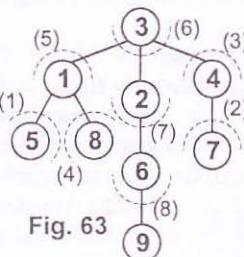


Fig. 63

Vectorii t și pt pentru arborele A_4 din figura 61 sunt:

(indice vector i)	1	2	3	4	5	6	7	8
Nod terminal ($t[i]$)	5	7	4	8	1	3	2	6
Părinte nod terminal ($pt[i]$)	1	4	3	1	3	2	6	9

Temă

- 1. Reprezentați arborele cu rădăcină A_5 cu vectorii t și pt .
- 2. Scrieți un program care să citească dintr-un fișier informații despre matricea de adiacență a unui arbore cu rădăcină (de pe primul rând – numărul de noduri n și eticheta rădăcinii r , iar de pe următoarele n rânduri – matricea de adiacență) și care să scrie vectorii t și pt ai arborelui într-un alt fișier.

Observații:

- Vectorii t și pt sunt aceiași pentru un arbore, indiferent de nodul rădăcină ales.
- La terminarea algoritmului, ultimul nod de care mai rămâne în arbore este nodul cu eticheta cea mai mare – n . Cunoscându-se această etichetă, în vectorul pt este suficient să se memoreze numai primele $n-2$ elemente, ultimul element având întotdeauna valoarea n .

3. Este suficient să se memoreze numai vectorul pt , deoarece din acest vector se poate construi vectorul t . Pentru a găsi algoritmul de construire a vectorului t , trebuie observat că eticheta j care se memorează în $t[i]$ corespunde unui nod care îndeplinește următoarele condiții:
- Nu a fost încă eliminat din arbore – deci nu se găsește printre etichetele $t[1], t[2], \dots, t[i-1]$.
 - Nu este părintele nici unui nod terminal care se va adăuga ulterior, deoarece va fi eliminat din arbore – deci nu se găsește printre etichetele $\text{pt}[i], \text{pt}[i+1], \dots, \text{pt}[n-1]$.
 - Are cea mai mică etichetă dintre nodurile care îndeplinesc condițiile (a) și (b).

Din aceste condiții rezultă că, pentru orice i ($1 \leq i \leq n-1$):

$$t[i] = \min\{k \mid k \in \{1, 2, \dots, n\} - \{t[1], t[2], \dots, t[i-1], \text{pt}[i], \text{pt}[i+1], \dots, \text{pt}[n-1]\}\}$$

Teorema 23

Numărul total de **arbori liberi** care se pot forma cu n noduri este n^{n-2} .

Demonstrație. Vectorul pt fiind același pentru un arbore cu rădăcină, indiferent de nodul rădăcină ales, el poate fi asociat și unui arbore liber. Vectorul pt are $n-2$ elemente. Notăm cu A mulțimea indicilor din vectorul pt și cu B mulțimea nodurilor arborelui liber. Fiecare arbore îl putem asocia o funcție $f: A \rightarrow B$ care asociază unui indice i din vectorul pt un nod j din arbore: $f(i)=j$. Invers, unei funcții f îl putem atașa un arbore. Notăm cu $a=\text{card}(A)=n-2$ și cu $b=\text{card}(B)=n$. Numărul de funcții $f: A \rightarrow B$ este egal cu $b^a = n^{n-2}$. Rezultă că numărul total de arbori care se pot forma cu n noduri este egal cu numărul de funcții f , adică n^{n-2} .

Temă

- Pentru arborele A_4 din figura 61 reconstituji vectorul t cu ajutorul vectorului pt .
- Scriți un program care să citească dintr-un fișier vectorul pt al unui arbore și care să construiască vectorul t . Salvați acest vector, în același fișier, pe rândul următor.

2.8.3.3. Algoritmi pentru parcurgerea unui arbore cu rădăcină

Parcurgerea unui arbore cu rădăcină se poate face prin:

- Algoritmul de **parcuregere în lățime**;
- Algoritmul de **parcuregere în adâncime**.

a) Algoritmul de parcuregere în lățime

Metoda: se prelucrăza mai întâi informația din nodul rădăcină, după care sunt prelucrate, de la stânga la dreapta, nodurile aflate pe primul nivel, apoi pe cel de al doilea etc.

Structura de date folosită este **coada** (c) în care se adaugă fiii nodului prelucrat.

Algoritmul pentru parcurgerea arborelui este următorul:

PAS1. Se inițializează coada cu nodul rădăcină r .

PAS2. Cât timp coada nu este vidă execută (prim \leq ultim):

PAS3. Este extras din coadă primul nod și este prelucrat.

PAS4. Sunt adăugați în coadă fii nodului prelucrat.

Exemplu. Coada de așteptare la parcurgerea arborelui A_6 din figura 64:

Pas	1	3	4	3	4	3	4	3	4	3	4	3	4	3
Nod curent	1	2	3	4	5	6	7	8	9	10	11	12	13	
Coada	1	-	2	3	3	4	4	5	6	5	6	6	7	8

Nodurile sunt parcuse în ordinea: 1, 2, 3, 4, 5, 6, 7, 8.

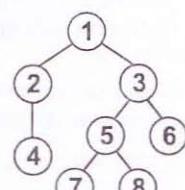


Fig. 64

Pentru implementarea algoritmului se folosesc subprogramele:

- funcția procedurală **citire** creează vectorul **t** (vectorul **tată**) prin preluarea informațiilor din fișier;
- funcția operand **rad** determină rădăcina **r** a arborelui;
- funcția procedurală **init** initializează coada de așteptare cu rădăcina;
- funcția operand **este_vida** testează coada de așteptare dacă este vidă;
- funcția procedurală **adauga** adaugă un nod la coada de așteptare;
- funcția procedurală **elimina** elimină nodul prelucrat din coada de așteptare;
- funcția procedurală **prelucrare** prelucrează primul nod din coadă: adaugă la coada de așteptare toți fiii acestui nod și apoi îl elimină din coada de așteptare;
- funcția procedurală **afisare** afișează nodurile arborelui în ordinea prelucrării.

```
#include <iostream.h>
int t[20],n,c[20],prim,ultim;
fstream f("arbore.txt",ios::in);
void citire() {f>>n; for (int i=1;i<=n;i++) f>>t[i]; f.close();}
int rad()
{for (int i=1;i<=n;i++) if (t[i]==0) return i;
 return 0;} //în caz de eroare - nu există nod rădăcină
void init (int k) {prim=ultim=1; c[ultim]=k;}
int este_vida() {return ultim<prim;}
void adaug(int i) {ultim++; c[ultim]=i;}
void elimina() {prim++;}
void prelucrare()
{int r=c[prim];
 for (int i=1;i<=n;i++) if (t[i]==r) adaug(i); elimina();}
void afisare() {for (int i=1;i<=n;i++) cout<<c[i]<<" ";}
void main() {int r; citire(); r=rad(); init(r);
 while (!este_vida()) prelucrare(); afisare();}
```

b) Algoritmul de parcurgere în adâncime

Metoda: pornind de la nodul rădăcină se prelucrează fiii unui nod de la stânga la dreapta, iar trecerea, de la nodul curent la fratele din dreapta, se face numai după ce au fost vizitați toți descendenții nodului curent, deci ai întregului subarbore dezvoltat din acesta. În funcție de ordinea relativă de prelucrare a nodului rădăcină și, respectiv, a subarborilor, există doi algoritmi:

- **Algoritmul de parcurgere în preordine**. Informația din nodul rădăcină este prelucrată înaintea informațiilor din celelalte noduri ale subarborilor. Implementarea arborelui se face prin **referințe descendente**.
- **Algoritmul de parcurgere în postordine**. Informația din nodul rădăcină este prelucrată după ce au fost prelucrate informațiile din toate celelalte noduri ale subarborilor. Implementarea arborelui se face prin **referințe ascendentе**.

Algoritmul de parcurgere în preordine

Structura de date folosită este **stiva** (**st**) în care informația este formată din perechi (nod tată, următorul fiu neprelucrat al acestuia). Inițial, se consideră că nodul prelucrat este nodul rădăcină. Notăm: **NC** = nod curent

NP = nod prelucrat (nodul care s-a vizitat)

PFNNC = primul fiu neprelucrat al nodului curent

PFNNP = primul fiu neprelucrat al nodului prelucrat

În vârful stivei se va memora perechea (NP, PFNNP).

Algoritmul pentru parcurgerea arborelui este următorul:

PAS1. Se inițializează stiva cu perechea (nodul rădăcină, primul fiu al rădăcinii).

PAS2. Cât timp stiva nu este vidă (mai există noduri în stivă) **execută**:

PAS3. Se extrag informații din vârful stivei despre:

- Nodul prelucrat devine nod curent $NC \leftarrow st[vf].NP$ (la prima parcurgere $NC \leftarrow 1$).
- Primul fiu neprelucrat al nodului prelucrat devine primul fiu neprelucrat al nodului curent $PFNNC \leftarrow st[vf].PFNNP$ (la prima parcurgere $PFNNC \leftarrow 3$).
- Primul fiu neprelucrat al nodului curent devine nod prelucrat $NP \leftarrow PFNNC$ (la prima parcurgere $NP \leftarrow 2$).
- Primul fiu neprelucrat al nodului prelucrat devine primul fiu neprelucrat al nodului prelucrat $PFNNP \leftarrow st[vf].PFNNP$ (la prima parcurgere $PFNNP \leftarrow 4$).

PAS4. Dacă nodul curent mai are și alți fii neprelucrați, **atunci** la stivă se adaugă perechea (NC, PFNNC). La prima parcurgere perechea este (1,3).

PAS5. Dacă nodul prelucrat nu este nod terminal și mai are și alți fii neprelucrați, **atunci** la stivă se adaugă perechea (NP, PFNNP). La prima parcurgere perechea este (2,4).

Exemplu. Stiva la parcurgerea arborelui A_6 din figura 64:

PAS	Stivă	NC	NP	PFNNC	PFNNP
1	(1, 2)		1		2
3	-	1	2	3	4
4	(1,3)	1	2	3	4
5	(1,3); (2, 4)	1	2	3	4
3	(1,3)	2	4	-	-
4	(1,3)	2	4	-	-
5	(1, 3)	2	4	-	-
3	-	1	3	-	5
4	-	1	3	-	5
5	(3, 5)	1	3	-	5
3	-	3	5	6	7
4	(3,6)	3	5	6	7
5	(3,6); (5, 7)	3	5	6	7
3	(3,6)	5	7	8	-
4	(3,6);(5,8)	5	7	8	-
5	(3,6); (5, 8)	5	7	8	-
3	(3,6)	5	8	-	-
4	(3,6)	5	8	-	-
5	(3, 6)	5	8	-	-
3	-	3	6	-	-
4	-	3	6	-	-
5	-	3	6	-	-

Nodurile sunt parcuse în ordinea: 1, 2, 4, 3, 5, 7, 8, 6.

Pentru implementarea algoritmului se folosesc subprogramele:

- funcția procedurală **citire** creează vectorul t (vectorul **tată**) prin preluarea informațiilor din fișier;
- funcția operand **rad** determină rădăcina r a arborelui;

- funcția operand `nodt` verifică dacă un nod este nod terminal;
- funcția procedurală `init` initializează stiva cu perechea (rădăcină, primul fiu al rădăcinii);
- funcția operand `este_vida` testează stiva dacă este vidă;
- funcția procedurală `adauga` adaugă un nod la stivă;
- funcția procedurală `elimina` elimină nodul din vârful stivei;
- funcția procedurală `prelucrare` prelucrează nodurile din vârful stivei astfel: nodul prelucrat devine nod curent, primul fiu neprelucrat al nodului curent devine nod prelucrat, afișează nodul prelucrat, elimină perechea de noduri din vârful stivei, dacă nodul curent mai are un fiu neprelucrat, adaugă la stivă perechea (**NC, PFNNC**), iar dacă nodul prelucrat nu este nod terminal și mai are un fiu neprelucrat, adaugă la stivă perechea (**NP, PFNNP**).

```
#include <fstream.h>
struct stiva {int NP, PFNNP;};
stiva st[20];
int t[20], n, vf;
fstream f("arbore.txt", ios::in);
int nodt(int x)
{for (int i=1; i<=n; i++) if (t[i]==x) return 0;
 return 1;}
void init (int r)
{int i=1, vf=1;
 while (i<=n && t[i]!=r) i++;
 st[vf].NP=r; st[vf].PFNNP=i;}
int este_vida() {return vf==0;}
void adaug(int x, int y) {vf++; st[vf].NP=x; st[vf].PFNNP=y;}
void elimin() {vf--;}
void prelucrare()
{int NC=st[vf].NP, PFNNC=st[vf].PFNNP, NP=PFNNC, PFNNP, i=PFNNC+1;
 cout<<NP<<" "; PFNNP=st[vf].PFNNP; elimin();
 while (i<=n && t[i]!=NC) i++;
 if (i<=n) {PFNNC=i; adaug(NC, PFNNC);}
 if (!nodt(NP))
 {i=PFNNP+1; while (i<=n && t[i]!=NP) i++;
 if (i<=n) {PFNNP=i; adaug(NP, PFNNP);}}
 void citire(){f>>n; for (int i=1; i<=n; i++) f>>t[i]; f.close();}
int rad()
{for (int i=1; i<=n; i++) if (t[i]==0) return i;
 return 0;}
void main()
{int r; citire(); r=rad(); init(r); cout<<"Nodurile vizitate: "<<r<<" ";
while (!este_vida()) prelucrare();}
```

Atenție

Prin parcurgerea în lățime a unui arbore cu rădăcină, prelucrarea informațiilor din noduri se face pe niveluri ierarhice, iar prin parcurgerea în adâncime în preordine a unui arbore, prelucrarea informațiilor din noduri se face după relațiile de subordonare.

Algoritmul de parcurgere în postordine

Nodului rădăcină î se va atribui cea mai mare etichetă. Prelucrarea se face prin parcurgerea simultană a celor doi vectori și căutarea nodurilor care aparțin aceluiași subarbore. Evidența nodurilor prelucrate este ținută prin intermediu vectorului `vizitat` (care este definit la fel ca și la algoritmii de parcurgere a grafurilor).

Algoritmul pentru parcugerea arborelui în postordine este următorul:

PAS1. Se initializează cu 0 elementele vectorului **vizitat**.

PAS2. Cât timp mai sunt noduri nevizitate **execută**:

PAS3. Se caută în vectorul **t** indicele **k** al primului nod terminal nevizitat.

PAS4. Se prelucrăză nodul **t[k]** și se declară vizitat.

PAS5. Pentru toate nodurile **t[i]** frate cu nodul **t[k]** **execută**

PAS6. Se prelucrăză nodul **t[i]** și se declară vizitat.

PAS7. Se prelucrăză părintele nodului **t[k]** (**pt[k]**) și se declară vizitat.

Pentru implementarea algoritmului se folosesc subprogramele:

- funcția procedurală **citire** creează vectorii **t** și **pt** prin prelucrarea informațiilor din fișier;
- funcția **terminat** verifică dacă au fost prelucrate toate nodurile arborelui;
- funcția procedurală **prelucrare** prelucrăză nodurile unui subarbore, pornind de la nodul terminal cu cea mai mică etichetă.

```
#include <fstream.h>
int t[20],pt[20],vizitat[20],n;
fstream f("arbore.txt",ios::in);
void citire()
{int i; f>>n; for (i=1;i<=n;i++) f>>t[i];
 for (i=1;i<=n;i++) f>>pt[i]; f.close();}
int terminat()
{for (int i=1;i<n;i++) if (!vizitat[i]) return 0;
 return 1;}
void prelucrare()
{int i,k;
 for (i=1;i<n && vizitat[t[i]]==0;i++)
 k=i; vizitat[t[k]]=1; cout<<t[k]<<" ";
 for (i=k+1;i<n;i++)
 if (pt[i]==pt[k]) {cout<<t[i]<<" "; vizitat[t[i]]=1;}
 cout<<pt[k]<<" "; vizitat[pt[k]]=1;}
void main() {citire(); cout<<"Nodurile vizitate sunt: ";
 while (!terminat()) prelucrare();}
```

Exemplu – În nodurile unui arbore se memorează eticheta nodului și un număr. La numărul memorat în fiecare nod se adună suma numerelor memorate în nodurile descendente. Să se afișeze noua valoare memorată în fiecare nod. Pentru a putea calcula aceste sume, parcugerea arborelui trebuie să se facă în postordine, cu ajutorul vectorilor **t** și **pt**. Numerele din fiecare nod vor fi memorate în vectorul **c**. Informațiile necesare se citesc dintr-un fișier text, astfel: de pe primul rând numărul de noduri **n**, iar de pe următoarele trei rânduri, cele **n** valori memorate în vectorii **t**, **pt** și **c**. Pe un rând, numerele sunt separate prin spațiu.

```
#include <fstream.h>
int t[20],pt[20],c[20],n;
fstream f("arbore.txt",ios::in);
void citire() //se citesc datele din fisier
{int i; f>>n; for (i=1;i<=n;i++) f>>t[i];
 for (i=1;i<=n;i++) f>>pt[i]; for (i=1;i<=n;i++) f>>c[i]; f.close();}
void prelucrare(){for (int i=1;i<=n;i++) c[pt[i]]=c[pt[i]]+c[t[i]];}
void afisare()
{for (i=1;i<=n;i++)
 cout<<"Nodul "<<i<<" are totalul "<<c[i]<<endl;}
void main() {citire(); prelucrare(); afisare();}
```

Temă

În nodurile unui arbore cu rădăcină sunt memorate numere naturale. Informațiile despre arbore se citesc dintr-un fișier text, astfel: de pe primul rând numărul de noduri ale arborelui, de pe rândul al doilea vectorul tată, și de pe al treilea rând, în ordinea etichetelor, numerele memorate în noduri. Scrieți un program care să calculeze suma numerelor pare memorate în arbore.

2.8.3.4. Aplicații practice

- Construieți arborele genealogic al familiei, care să prezinte filiația (descendenții direcți ai unei persoane). Se va lua ca nod rădăcină unul dintre stră-străbunici. În fiecare nod se va memora numele unei persoane. Informațiile despre arbore se citesc dintr-un fișier text, astfel: de pe primul rând numărul de noduri ale arborelui, de pe rândul următor vectorul tată, și de pe al treilea rând, în ordinea etichetelor, numele memorate în noduri. Scrieți un program care să furnizeze următoare informații:
 - Pentru un nume de persoană, citit de la tastatură, să se afișeze descendenții direcți și căță copii are fiecare.
 - Pentru două nume de persoană, citite de la tastatură, să se afișeze strămoșul comun.
- O fabrică de confecții este formată din mai multe compartimente. Între compartimente există fie relații de subordonare, fie relații de colaborare (relațiile de colaborare se stabilesc pe același nivel ierarhic). În figura 65 este prezentată organigrama firmei. Fiecare compartiment are următoarele atribute: numele compartimentului, numele persoanei care conduce acel compartiment și numărul de angajați.

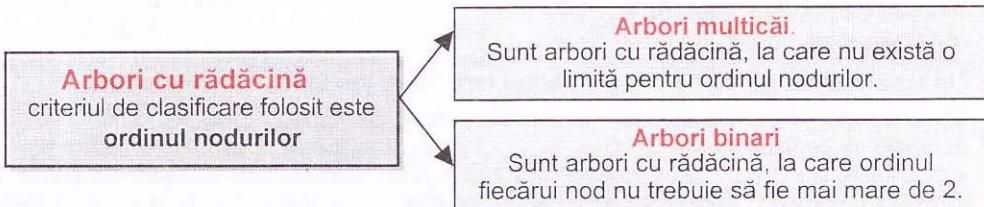


Fig. 65

- Afișați lista angajaților care au funcții de conducere (numele compartimentului și numele persoanei) în două moduri:
 - punând în evidență relațiile de subordonare;
 - grupând persoanele de pe același nivel ierarhic.
- Afișați lista angajaților care au funcții de conducere și numărul total de angajați pe care îi are fiecare dintre ei în subordine (o persoană cu funcție de conducere are în subordine angajații din compartimentul pe care îl conduce și din toate compartimentele subordonate acestuia).
- O rețea de distribuție a produselor unei firme are o organizare piramidală în care fiecare dintre agenții comerciali are în subordine alți agenții comerciali. În funcție de vânzări, fiecare agent comercial primește un punctaj. Un agent comercial care are în subordine și alți agenții comerciali cumulează la punctajul propriu și punctajele agenților comerciali din subordine. Scrieți un program care să afișeze următoarele informații:
 - punctajul total al rețelei;
 - cum este subordonatul direct al unui agent comercial (al cărui nume se precizează de la tastatură) care are punctajul cel mai mare și care este acest punctaj.

4. Angajații unei firme dispun de o sumă pentru prime distribuită astfel: directorul general își stabilește prima și restul sumei o distribuie în mod egal subordonatilor lui direcți. Orice subordonat care a primit o sumă își stabilește singur prima (o parte sau întreaga sumă primită) restul sumei fiind distribuită în mod egal subordonaților săi direcți, iar angajații care nu au subordonați rețin întreaga sumă pe care o primesc. Știind că orice subordonat are un singur șef, scrieți un program de distribuire a primelor și afișați lista angajaților și primele primite. Pentru prima pe care o stabilește pentru sine un angajat, folosiți generatorul de numere aleatoare, care va genera un număr între 0 și suma primită pentru prime – pentru a o distribui.

(Sesiunea august Bacalaureat 2003 – adaptată)



2.8.4. Arboarele binar

2.8.4.1. Definiția arborelui binar

Se numește **arbore binar** un **arbore cu rădăcină pozitional** care are proprietatea că **fiecare nod are cel mult doi descendenți direcți** (succesori).

Terminologie:

→ Cei doi succesiști ai unui nod (dacă există) se numesc **succesor stâng** (**subarbore stâng**) și **succesor drept** (**subarbore drept**) – arborele A₇ din figura 66. A₇

Caracteristici:

- Deoarece, oricare ar fi un nod al arborelui, el nu are mai mult de doi descendenți direcți, **ordinul unui nod** dintr-un arbore binar poate fi 0 (nod terminal), 1 (unul dintre subbarbore este vid) sau 2.
- Definiția arborelui binar este o **definiție recursivă**. Există un nod privilegiat – numit **nod rădăcină**, iar celelalte noduri (dacă există) sunt repartizate în **două grupuri disjuncte** și, fiecare dintre aceste grupuri formează, la rândul său, un arbore binar.

Observație. Arboarele binar fiind un arbore cu rădăcină pozitional, se face diferență între succesișorul stâng și succesișorul drept, adică, dacă un nod are un singur descendenter, trebuie să se precizeze care dintre descendenteri este. În figura 67 sunt prezentate doi arbori binari A_{b1} și A_{b2} care sunt diferenți, chiar dacă au două noduri și rădăcină în nodul cu eticheta A. În arborele A_{b1} succesișorul stâng este un nod (B), iar succesișorul drept este arborele vid, iar în arborele A_{b2} succesișorul stâng este arborele vid, iar succesișorul drept este un nod (B).

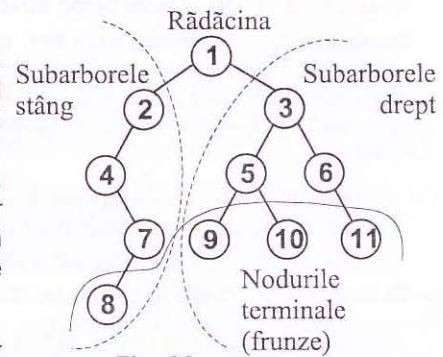


Fig. 66

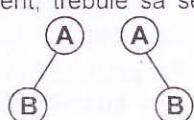


Fig. 67

Cu 3 noduri, etichetate cu A, B și C, se obțin 30 de arbori binari diferenți. Există trei moduri de a alege rădăcina. Pentru un nod rădăcină ales, se pot obține zece arbori binari diferenți – în total se vor obține $3 \times 10 = 30$ arbori binari diferenți. În figura 68 sunt prezentate 6 dintre arborii binari care au rădăcina în nodul A.

Se numește arbore binar strict un arbore care are proprietatea că fiecare nod, cu excepția nodurilor terminale, are exact doi descendenți (succesori).

A₈ Exemplu – Arboarele binar A₈ din figura 69

Propoziția 17

Un arbore binar strict, care are **n** noduri terminale, are în total **$2 \times n - 1$** noduri.

Demonstrație – Notăm cu k numărul de nivele din arbore, cu $x_k, x_{k-1}, \dots, x_2, x_1$, numărul de noduri terminale de pe fiecare nivel, și cu $y_k, y_{k-1}, \dots, y_2, y_1$, numărul de noduri de pe fiecare nivel (cu excepția nivelului 0, pe care se găsește rădăcina) și cu N – numărul total de noduri din arbore ($N = y_k + y_{k-1} + \dots + y_2 + y_1 + 1$), cu n – numărul total de noduri terminale ($n = x_k + x_{k-1} + \dots + x_2 + x_1$). Nivelul k conține numai noduri terminale, și $y_k = x_k$. Pentru fiecare alt nivel există relația: $y_i = y_{i+1}/2 + x_i$. Arboarele fiind strict, $y_{i+1}/2$ este un număr întreg, deoarece pe fiecare nivel există un număr par de noduri. Adunând relațiile, obținem:

$$\begin{aligned} y_k &= x_k \\ y_{k-1} &= y_k/2 + x_{k-1} \\ y_{k-2} &= y_{k-1}/2 + x_{k-2} \\ \cdots &\cdots \\ y_2 &= y_3/2 + x_2 \\ y_1 &= y_2/2 + x_1 \end{aligned}$$

Rezultă că: $N = 1 + y_k + y_{k-1} + \dots + y_2 + y_1 + y_0 = 1 + 2 \times (x_k + x_{k-1} + \dots + x_2 + x_1) = 2 \times n + 1$

Propoziția 16. Un arbore binar strict are un număr impar de noduri.

Demonstrație – Pe fiecare nivel $k+1$, pentru fiecare nod de pe nivelul k , există câte doi descendenți sau nici un descendant. Rezultă că pe fiecare nivel, cu excepția nivelului 0, există un număr par de noduri. Numărul total de noduri va fi impar, deoarece la aceste noduri se adaugă și nodul rădăcină.

Se numește arbore binar echilibrat un arbore binar care are proprietatea că diferența dintre înălțimile celor doi subarborei ai oricărui nod este cel mult 1.

A₉ Exemplu – Arboarele A₉ din figura 70.

Se numește arbore binar perfect echilibrat un arbore binar care are proprietatea că diferența dintre numărul nodurilor celor doi subarborei ai oricărui nod este cel mult 1.

A₁₀ Exemplu – Arboarele A₁₀ din figura 71

Proprietate. Un arbore binar cu **n** noduri este perfect echilibrat dacă subarborele stâng are $[n/2]$ noduri, iar subarborele drept are $n-[n/2]-1$ noduri.

Se numește arbore binar complet un arbore binar strict care are toate nodurile terminale pe același nivel.

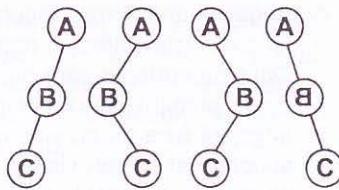


Fig. 68

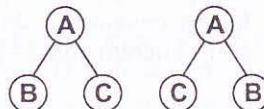


Fig. 69

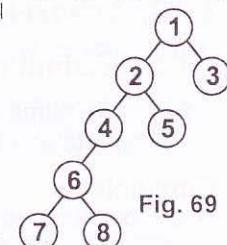


Fig. 70

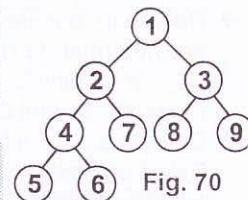


Fig. 71

A₁₁ Exemplu – Arboarele A₁₁ din figura 72

Propoziția 19. Un arbore binar complet, care are n noduri terminale, are în total $2 \times n - 1$ noduri.

Demonstrație – Folosind principiul inducției matematice, demonstrăm că, într-un arbore binar complet, pe nivelul k sunt 2^k noduri (se notează cu P_k propoziția i).

P_0 – Pe nivelul 0 există un singur nod (nodul rădăcină), adică $2^0 = 1$ nod.

P_1 – Pe nivelul 1 există două noduri, adică $2^1 = 2$ noduri.

P_k – Pe nivelul k există 2^k noduri..

P_{k+1} – Considerând propoziția P_k adevărată, trebuie să demonstrează că pe nivelul $k+1$ există 2^{k+1} noduri. Dacă nivelul $k+1$ aparține arborelui, atunci pe acest nivel există câte doi descendenți pentru fiecare nod de pe nivelul k , în total $2 \times 2^k = 2^{k+1}$ noduri.

Considerând că arborele are k niveluri și că numărul de noduri de pe nivelul k este n , numărul total de noduri din arbore se obține adunând nodurile de pe cele k niveluri:

$$2^0 + 2^1 + 2^2 + \dots + 2^k = 2^{k+1} - 1 = 2 \times (2^k) - 1 = 2 \times n - 1$$

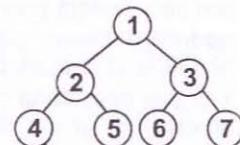


Fig. 72

Se numește **arbore binar aproape complet** un arbore binar complet până la penultimul nivel, la care completarea cu noduri, pe ultimul nivel, se face de la stânga la dreapta.

Exemplu – Arborele A_{12} din figura 73

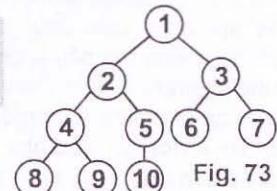


Fig. 73

Studiu de caz

Scop: identificarea colecțiilor de date care pot fi reprezentate prin arbori binari.

Enunțul problemei 1. Trebuie să se organizeze un campionat de meciuri de baschet între echipele mai multor licee. Numărul de echipe este n . În fiecare duminică se organizează un meci. Echipele care vor juca în prima etapă se trag la sorți. Să se realizeze planificarea meciurilor și să se determine numărul de duminici în care se va desfășura campionatul (figura 74).

Campionatul are o structură ierarhică, iar relațiile dintre componente pot fi reprezentate cu ajutorul unui arbore binar, în care cele două echipe care joacă sunt nodurile. Nodurile terminale sunt echipele liceelor, iar celelalte noduri reprezintă câte o echipă care a câștigat în meciul etapei anterioare. Rădăcina arborelui este echipa care câștigă finala. Deoarece o echipă câștigătoare se desemnează în urma unui meci jucat, nodurile care nu sunt terminale reprezintă meciuri jucate, iar nodul rădăcină corespunde meciului final. Arcele reprezintă relația de participare a unei echipe la un meci. Arborele meciurilor este un **arbore binar strict** (la un meci participă două echipe). Dacă arborele meciurilor este un **arbore binar complet** și la campionat participă n echipe, vor fi n noduri terminale și arborele va avea $2 \times n - 1$ noduri. Numărul de meciuri care se vor juca este dat de diferența dintre numărul total de noduri ale arborelui și numărul de noduri terminale, adică $n - 1$ meciuri, iar campionatul se va desfășura în $n - 1$ duminici.

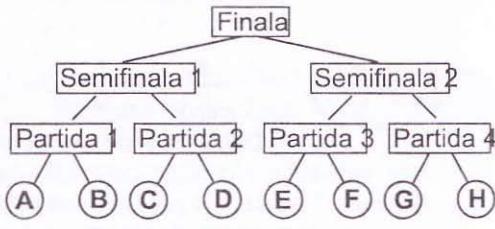


Fig. 74

Enunțul problemei 2. Să se reprezinte modul în care se evaluatează o expresie aritmetică.

O expresie aritmetică este formată din operanți (constante și variabile), operatori aritmetici (+, -, /, * și ^ – pentru ridicarea la putere) și paranteze. În evaluarea expresiei aritmetice

tice se respectă prioritatea operatorilor aritmetici. **Operatorii aritmetici sunt operatori binari care nu sunt toți comutativi** (operatorii pentru scădere, împărțire și ridicare la putere nu sunt comutativi). Dacă notăm operanții cu x și y și operatorul cu op , atunci expresiei $E = x \ op \ y$ putem să-i asociem arborele binar din figura 75, în care rădăcina este operatorul op , subarborele stâng este primul operand – x , iar subarborele drept este al doilea operand – y . Dacă expresia conține un singur operand, acestuia i se poate asocia un arbore binar cu un singur nod – nodul rădăcină – în care se memorează operandul. Pentru a putea reprezenta o expresie care este formată cu un operator unar (de exemplu, $E=-x$), vom transforma expresia astfel încât să conțină un operator binar (în exemplu, $E=0-x$).

Expresia are o structură ierarhică, iar relațiile dintre componente pot fi reprezentate cu ajutorul unui arbore binar, în care operanții și operatorii sunt nodurile. Nodurile terminale sunt operanții, iar celelalte noduri reprezintă operatorii. Rădăcina arborelui este operatorul care se evaluează ultimul. Subarborele stâng și subarborele drept ai rădăcinii reprezintă expresiile cărora li se aplică ultimul operand. Rădăcina fiecărui subarbore va reprezenta operatorul care se aplică pe cele două expresii reprezentate prin subarborele stâng și drept ai săi. Arcele reprezintă relația de participare a unei expresii la operatorul din nodul părinte: ca prim operand sau ca al doilea operand. Arborele expresiei este un **arbore binar strict** (un operator leagă doi operanți). Dacă expresia conține n operanți, arborele va avea n noduri terminale. Fiind un arbore binar strict, va avea $2n-1$ noduri. Numărul de operatori este dat de diferența dintre numărul total de noduri ale arborelui și numărul de noduri terminale, adică $n-1$ operatori. Arborele expresiei nu este un **arbore echilibrat**. Arborele binar din figura 76 este arborele expresiei aritmetice:

$$E = 5 \times \frac{(-x^2 + 4 \times y)}{z + y} + \frac{x}{y}$$

Tema



Desenați 5 arbori binari cu trei noduri (etichetate cu A, B și C) diferiți, altii decât cei din figura 69.

1. Desenați arborele genealogic al strămoșilor unei persoane până la nivelul stră-străbunici. Rădăcina este persoana pentru care se întocmește arborele genealogic. Fiecare nod are doi descendenti: mama și tata. Ce tip de arbore binar este? Comparați acest arbore cu arborele genealogic al descendenților (arborele filiației).
2. Desenați arborele campionatului de baschet pentru 7 echipe. Ce fel de arbore binar este?

3. Desenați arborele binar al expresiei $\frac{a^2 + b}{c - d} + 7 \times (a - b)$

2.8.4.2. Implementarea arborelui binar

Implementarea structurilor de date de tip arbore binar se poate face:

- a. **static** – folosind vectori;
- b. **dinamic** – folosind pointeri.

a) Implementarea statică a arborelui binar

Există două metode de implementare statică a arborilor binari (se folosesc arborile binar A₁₃ din figura 77).

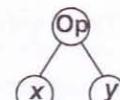


Fig. 75

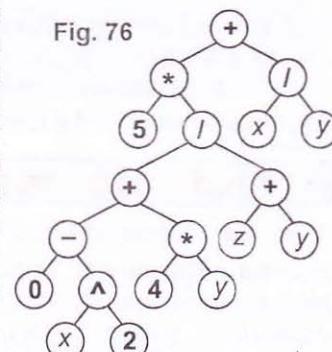


Fig. 76

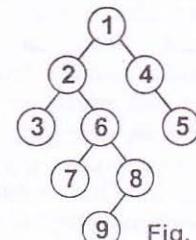


Fig. 77

1. Folosind doi vectori în care **se memorează cei doi succesorii ai unui nod**

- vectorul **st** – în elementul i se memorează eticheta nodului succesor stâng al nodului i ;
- vectorul **dr** – în elementul i se memorează eticheta nodului succesor drept al nodului i .

Dacă nodul i nu are succesor stâng, respectiv drept, elementul din vectorul **st**, respectiv **dr**, va avea valoarea 0.

Pentru arborele A_{13} din figura 77 cei doi vectori sunt:

Nodul i	1	2	3	4	5	6	7	8	9
st[i]	2	3	0	0	0	7	0	9	0
dr[i]	4	6	0	5	0	8	0	0	0

Observație. Din cei doi vectori puteți obține următoarele informații:

- eticheta nodului rădăcină r – nodul i pentru care, oricare ar fi indicele j , $st[j] \neq i$ și $dr[j] \neq i$ (nodul a cărui etichetă nu există nici în vectorul **st**, nici în vectorul **dr**);
- etichetele nodurilor terminale – nodurile i pentru care $st[i]+dr[i]=0$;
- eticheta părintelui unui nod i – indicele j pentru care $st[j]=i$ sau $dr[j]=i$;
- etichetele fiilor unui nod i – **st[i]** și **dr[i]** (dacă sunt diferenți de 0);
- eticheta fratelui unui nod i – **st[j]** pentru $dr[j]=i$ sau **dr[j]** pentru $st[j]=i$.

2. Folosind doi vectori în care **se memorează filiația nodurilor**

- vectorul **tata** – în elementul i se memorează numărul de ordine al nodului predecesor (părinte) al nodului i ;
- vectorul **fii** (în elementul i se memorează ce fel de succesor al părintelui este; dacă **fii[i]=-1**, atunci nodul i este succesorul stâng al părintelui său, iar dacă **fii[i]=1**, atunci nodul i este succesorul drept al părintelui său).

Dacă nodul i este nodul rădăcină, elementul din vectorul **tata**, respectiv **fii**, va avea valoarea 0.

Pentru arborele A_{13} din figura 77 cei doi vectori sunt:

Nodul i	1	2	3	4	5	6	7	8	9
tata[i]	0	1	2	1	4	2	6	6	8
fii[i]	0	-1	-1	1	1	1	1	-1	-1

Observație. Din vectorul tată puteți obține următoarele informații:

- eticheta nodului rădăcină r – indicele i pentru care **tata[i]=0**;
- etichetele nodurilor terminale – nodurile i a căror etichetă nu există în vectorul **tata**;
- eticheta părintelui unui nod i – **tata[i]**;
- etichetele fiilor unui nod i – fiul stâng: indicele j pentru care **tata[j]=i** și **fii[j]=-1**, iar fiul drept: indicele j pentru care **tata[j]=i** și **fii[j]=1**;
- eticheta fratelui unui nod i – indicele j pentru care **tata[i]=tata[j]**.

Observație. Datorită definiției arborilor binari, algoritmii utilizati pentru prelucrarea lor pot folosi **tehnica recursivității** (definiția recursivă a arborilor binari) și **strategia divide et impera** (fiecare nod nu are decât doi descendenți – prelucrarea unui nod se descompune în două subprobleme: prelucrarea subarborelui stâng și prelucrarea subarborelui drept, urmată de compunerea celor două soluții).

Exemple – dacă pentru arborele binar se folosește implementarea statică, pentru a construi algoritmi recursivi cu strategia divide et impera se utilizează vectorii **st** și **dr**.

Înălțimea arborelui binar

```
int max(int x, int y)
{if (x>y) return x; else return y;}
int h(int i)
{if (st[i]==0 && dr[i]==0) return 0;
 else
    return 1+max(h(st[i]),h(dr[i]));}
void main()
{int r; ... cout<<h(r); ... }
```

Temă

1. Scrieți un program care să furnizeze următoarele informații despre un arbore binar: nodul rădăcină, nodurile terminale, nodurile cu exact un fiu, nodurile cu exact doi fii, nodurile frați și înălțimea. Informațiile despre arbore (numărul de noduri și cei doi vectori) se vor citi dintr-un fișier. Arborele este implementat:

- cu vectorii în care se memorează cei doi succesi ai unui nod;
 - cu vectorii în care se memorează filiația nodurilor.
2. Scrieți un program care citește dintr-un fișier numărul de noduri și vectorii **st** și **dr** ai unui arbore binar. Să se reprezinte arborele prin vectorii **tata** și **fii**.
3. Scrieți un program care citește dintr-un fișier numărul de noduri și vectorii **tata** și **fii** ai unui arbore binar. Să se reprezinte arborele prin vectorii **st** și **dr**.
4. Scrieți un program care citește dintr-un fișier numărul de noduri și vectorii **st** și **dr** ai unui arbore binar. Să se verifice dacă arborele binar este un arbore binar strict.
5. Scrieți un program care citește dintr-un fișier numărul de noduri și vectorii **st** și **dr** ai unui arbore binar. Să se verifice dacă este un arbore binar perfect echilibrat sau un arbore binar echilibrat.

b) Implementarea dinamică a arborelui binar

Se face prin definirea unui tip de dată pentru un nod din arbore (tipul **nod**) și a adresei unui nod (un pointer către tipul de dată nod – **nod***). Tipul de dată este definit ca o înregistrare care conține 3 categorii de câmpuri:

- **informația utilă** – poate fi compusă din mai multe câmpuri (<tip> **info**);
- **adresa subarborelui stâng** – (**nod *s** → **s** este adresa rădăcinii subarborelui stâng);
- **adresa subarborelui drept** – (**nod *d** → **d** este adresa rădăcinii subarborelui drept).

```
struct nod {<tip> info;;
            nod *s,*d;};
nod *r; //r=adresa rădăcinii
```

Trebuie precizate în fiecare moment pozițiile a trei componente: rădăcina (*r) și cei doi subbarbri (*s și *d).

Regulă. Întotdeauna este reprezentat subarborele stâng și apoi subarborele drept.

Dacă un nod nu are un succesor, succesorul va fi considerat arborele vid.

Se numește **arbore vid** un arbore care are adresa **NULL**.

Numărul de „frunze“ din arborele binar

```
int frunza(int i)
{return st[i]+dr[i]==0;}
int nr_fr(int i)
{if (frunza(i)) return 1;
 else
    if (st[i]==0)
        return nr_frunze(dr[i]);
    else
        if (dr[i]==0)
            return nr_frunze(st[i]);
        else
            return nr_fr(st[i])+nr_fr(dr[i]);}
void main()
{int r; ... cout<<nr_fr(r); ... }
```

Exemplul 1 – Pentru un arbore binar pe care-l construim pentru campionatul de meciuri de baschet între echipele mai multor licee, informația utilă poate fi formată din mai multe câmpuri: numele echipei (`n_e`), numele liceului (`n_l`) și eticheta nodului (`nr`):

```
struct nod {char n_e[15],n_l[15];
            int nr;
            nod *s,*d;} *r;
```

Exemplul 2 – Pentru exemplele următoare, informația utilă va fi formată numai din eticheta nodului:

```
struct nod {int nr;
            nod *s,*d;};
nod *r;
```

În figura 78 este prezentată implementarea dinamică a arborelui A_{13} din figura 77.

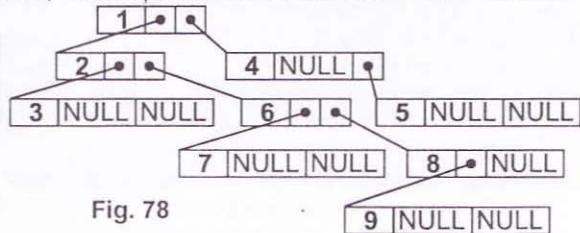


Fig. 78

Algoritmul pentru crearea unui arbore binar

Algoritmul pentru crearea unui arbore binar implementat dinamic folosește strategia divide et impera prin descompunerea problemei în trei subprobleme:

1. crearea nodului rădăcină (cazul de bază);
2. crearea subarborelui stâng;
3. crearea subarborelui drept.

Descompunerea problemei continuă până când subarborele care se creează este arborele vid. Combinarea soluțiilor se face prin legarea nodului rădăcină de cei doi subarbore.

Dacă datele se citesc de la tastatură, pentru a evidenția că **nodul care se creează este rădăcina unui arbore vid**, în **câmpul pentru etichetă se introduce valoarea 0** (care marchează lipsa de informații pentru acel nod). De exemplu, pentru a crea arborele din figura 79, se introduc în ordine, de la tastură, etichetele: 1 2 3 0 0 6 7 0 0 8 9 0 0 0 4 0 5 0 0. Algoritmul pentru crearea arborelui binar este:

PAS1. Se citește informația utilă din nod.

PAS2. Dacă nodul nu conține informație utilă, atunci se creează arborele vid; altfel:

PAS3. Se creează un nod rădăcină prin alocarea unei zone de memorie.

PAS4. Se atribuie, câmpului cu informație din nod, informația citită.

PAS5. Se creează subarborele stâng.

PAS6. Se creează subarborele drept.

Subprogramul care creează un arbore binar poate fi implementat ca funcție procedurală sau ca funcție operand.

Funcție procedurală

```
void creare(nod *&r)
{int n;
cout<<"Eticheta nod: "; cin>>n;
if (n==0) r=NULL;
else {r = new nod;
       r->nr=n;
       creare(r->s);
       creare(r->d);}}
```

Funcție operand

```
nod * creare()
{int n; nod *r;
cout<<"Eticheta nod: "; cin>>n;
if (n==0) return NULL;
else {r = new nod;
       r->nr=n;
       r->s=creare();
       r->d=creare();
       return r;}}
```

```
void main()
{creare(r);
 cout<<"Radacina= "<<r->nr;... }
```

Exemple – Funcții, implementate recursiv folosind strategia **divide et impera, prin care se prelucrează arbori binari.**

Înălțimea arborelui binar

```
int max(int x, int y)
{if (x>y) return x;
 else return y;}
int h(nod *r)
{if (r==NULL) return 0;
 else
    return 1+max(h(r->s),h(r->d));}
```

Copierea unui arbore binar

```
nod *comb(int n,nod *s,nod *d)
{nod *c; c = new nod; c->nr=n;
 c->s=s; c->d=d; return c;}
nod *copie(nod *c)
{if (c==NULL) return NULL;
 else return
comb(c->nr,copie(c->s),copie(c->d));}
void main()
{rl=creare(); r2=copie(rl); ... }
```

Afișarea etichetelor de pe un nivel precizat

```
void nivel(nod *r,int i,int k)
{if (r!=NULL)
 {if (i==k) cout<<r->nr<<" ";
  nivel(r->s,i+1,k);
  nivel(r->d,i+1,k);}}
void main()
{int k; r=creare();
 cout<<"Nivelul: "; cin>>k;
 nivel(r,0,k);}
```

```
void main()
{r=creare();
 cout<<"Radacina= "<<r->nr;... }
```

Numărul de „frunze“ ale arborelui binar

```
int frunza(nod *r)
{return r->s==NULL && r->d==NULL;}
int nr_fr(nod *r)
{if (r==NULL) return 0;
 else if (frunza(r)) return 1;
 else
    return nr_fr(r->s)+nr_fr(r->d); }
```

Compararea a doi arbori binari

```
int egal(nod *r1,nod *r2)
{if (r1==NULL) return r2==NULL;
 else if (r2==NULL) return 0;
 else return
    r1->nr==r2->nr &&
    egal(r1->s,r2->s) &&
    egal(r1->d,r2->d);}
void main()
{... if (egal(r1,r2)) ... }
```

Verificarea existenței unui nod cu eticheta precizată

```
int caut(nod *r,int k)
{if (r==NULL) return 0;
 else
 {if (r->nr==k) return 1;
  else
    return caut(r->s,k)||caut(r->d,k);}}
void main()
{int k; r=creare();
 cout<<"Nodul: "; cin>>k;
 if (caut(r,k))
    cout<<"A fost gasit";
 else cout<<"Nu a fost gasit";}
```

2.8.4.3. Algoritmi pentru parcurgerea unui arbore binar

Parcurgerea unui arbore binar se poate face prin:

- algoritmul de **parcurgere în lățime** (la fel ca la grafuri);
- algoritmi de **parcurgere în adâncime** (specifici arborilor binari):
 - algoritmul **RSD** (traversarea în preordine);
 - algoritmul **SRD** (traversarea în inordine);
 - algoritmul **SDR** (traversarea în postordine).

Observație. Algoritmii pentru parcurgerea unui arbore binar pot fi folosiți pentru a prelucra informațiile din noduri.

Algoritmul RSD

Metoda: se prelucrează rădăcina, subarborele stâng, subarborele drept (pentru arborele A₁₃ din figura 77 – 1 2 3 6 7 8 9 4 5).

Implementarea statică

```
void rsd(int i)
{cout<<i<<" ";
 if (st[i]!=0) rsd(st[i]);
 if (dr[i]!=0) rsd(dr[i]);}
void main()
{ ...rsd(r); cout<<endl; ...}
```

Implementarea dinamică

```
void rsd(nod *r)
{if (r!=NULL)
 {cout<<r->nr<<" ";
 rsd(r->s);
 rsd(r->d);}}
void main()
{ ...rsd(r); cout<<endl; ...}
```

Algoritmul SRD

Metoda: se prelucrează subarborele stâng, rădăcina, subarborele drept (pentru arborele A₁₃ din figura 77 – 3 2 7 6 9 8 1 4 5).

Implementarea statică

```
void srd(int i)
{if (st[i]!=0) srd(st[i]);
 cout<<i<<" ";
 if (dr[i]!=0) srd(dr[i]);}
void main()
{ ...srd(r); cout<<endl; ...}
```

Implementarea dinamică

```
void srd(nod *r)
{if (r!=NULL)
 {srd(r->s);
 cout<<r->nr<<" ";
 srd(r->d);}}
void main()
{ ...srd(r); cout<<endl; ...}
```

Algoritmul SDR

Metoda: se prelucrează subarborele stâng, subarborele drept, rădăcina (pentru arborele A₁₃ din figura 77 – 3 7 9 8 6 2 5 4 1):

Implementarea statică

```
void sdr(int i)
{if (st[i]!=0) srd(st[i]);
 if (dr[i]!=0) srd(dr[i]);
 cout<<i<<" ;"}
void main()
{ ...sdr(r); cout<<endl; ...}
```

Implementarea dinamică

```
void sdr(nod *r)
{if (r!=NULL)
 {sdr(r->s);
 sdr(r->d);
 cout<<r->nr<<" ;"}}
void main()
{ ...sdr(r); cout<<endl; ...}
```

Observație. Informațiile din nodurile arborelui binar, după prelucrare, pot fi salvate într-un fișier text. La o nouă execuție a programului ele permit restaurarea arborelui pentru o nouă prelucrare.

Exemplu – pentru arborele creat anterior, etichetele nodurilor sunt salvate într-un fișier text (sunt scrise pe un rând, despărțite prin spațiu) și la o nouă execuție a programului sunt readuse în memoria internă:

Salvarea arborelui în fișierul text

```
fstream f("arbore.txt",ios::out);
void salveare(nod *r)
{if (r!=NULL)
 {f<<r->nr<<" ";
 rsd(r->s);
 rsd(r->d);}
 else f<<0<<" ";
```

Restaurarea arborelui din fișierul text

```
fstream g("arbore.txt",ios::in);
nod *restaurare()
{int n; nod *r; g>>n;
 if (n==0) return NULL;
 else {r = new nod;
 r->nr=n;
 r->s=restaurare();
 r->d=restaurare();}}
```

```
void main()
{ ...salvare(r); ... }
```

```
r->d=restaurare();
return r;}}
```

```
void main()
{ ...r=restuarare(r); ... }
```

Temă

1. Scrieți câte un subprogram pentru următoarele metode de parcurgere a arborilor binari: **RDS** (rădăcină, subarbore drept, subarbore stâng), **DRS** (subarbore drept, rădăcină, subarbore stâng) și **DSR** (subarbore drept, subarbore stâng, rădăcină).
2. Scrieți un program care citește succesiunea nodurilor afișate prin parcurgerea SDR și prin parcurgerea RSD și afișează etichetele nodurilor prin parcurgerea RSD.
3. Scrieți o funcție iterativă pentru metoda de parcurgere RSD a arborilor binari.
4. Scrieți o funcție care afișează numărul nivelului cu cele mai multe frunze.
5. Scrieți o funcție care să verifice dacă un arbore binar este perfect echilibrat.
6. Scrieți un program care să furnizeze următoarele informații despre un arbore binar: nodul rădăcină, nodurile terminale, nodurile cu exact un fiu, nodurile cu exact doi fi, nodurile frați și înălțimea. Arborele binar este implementat dinamic și crearea lui se face prin introducerea informațiilor din noduri de la tastatură.
7. Scrieți un program care creează un arbore binar implementat dinamic prin citirea etichetelor dintr-un fișier text (etichetele sunt scrise pe un rând, despărțite prin spațiu, în ordinea în care ar fi fost introduse de la tastatură) și care verifică dacă arborele binar este un arbore binar strict.
8. În nodurile unui arbore binar sunt memorate numere întregi care nu sunt distințe. Scrieți un program care creează un arbore binar implementat dinamic prin citirea numerelor dintr-un fișier text (numerele sunt scrise pe un rând, despărțite prin spațiu, în ordinea în care ar fi fost introduse de la tastatură) și care creează o listă dublu înălțuită care să conțină numai numerele distințe și, pentru fiecare număr, de câte ori apare în arbore.
9. În nodurile unui arbore binar sunt memorate numere întregi care nu sunt distințe. Scrieți un program care creează un arbore binar implementat dinamic prin citirea numerelor dintr-un fișier text și care creează o listă simplă, ordonată descrescător, care să conțină numai numerele distințe.
10. În nodurile unui arbore binar sunt memorate numere naturale. Scrieți un program care să realizeze următoarele:
 - a. Creează arboarele binar (implementat dinamic) prin introducerea numerelor de la tastatură.
 - b. Calculează suma numerelor pare din nodurile terminale.
 - c. Calculează suma numerelor pozitive din nodurile care au exact doi succesi.
 - d. Determină numărul cu valoarea cea mai mare și de câte ori apare în nodurile arborelui.
 - e. Creează o listă simplu înălțuită cu numerele din nodurile arborelui care au ultima cifră 5 sau 3.
 - f. Afișează pe ecran numerele de pe un nivel **k** (**k** se citește de la tastatură).
 - g. Salvează, într-un fișier text, numerele de pe fiecare nivel (numerele de pe același nivel vor fi scrise pe un rând, separate prin spațiu).
 - h. Creează un arbore cu aceeași structură arborescentă ca cel creat la punctul (a) dar care diferă de el prin informația din noduri: dacă în primul arbore numărul este par, în al doilea arbore este înlocuit cu suma cifrelor sale, iar dacă în primul arbore numărul este impar, în al doilea arbore este înlocuit cu inversul lui.

2.8.4.4. Aplicații practice

- Construiți arborele genealogic al strămoșilor familiei până la nivelul stră-străbunici (părinții direcți ai fiecărei persoane). Se va lua ca nod rădăcină persoana pentru care se întocmește arborele. În fiecare nod se va memora numele unei persoane (mama sau tatăl persoanei din nodul părinte). Informațiile despre arbore se citesc dintr-un fișier text, astfel: de pe primul rând, numărul de noduri ale arborelui, de pe următoarele două rânduri vectorii **st** și **dr**, și de pe al patrulea rând, în ordinea etichetelor, numele memorate în noduri. Scrieți un program care, pentru un nume citit de la tastatură, să afișeze numele părinților și numele buniciilor persoanei respective.
- Construiți arborele campionatului de meciuri de baschet. În fiecare nod se vor păstra următoarele informații: eticheta nodului, numele celor două echipe, scorul fiecărei echipe și data la care s-a jucat meciul. În nodurile terminale vor fi păstrate numai numele echipelor. Scrieți un program care să asigure următoarea funcție: după fiecare meci, se va completa informația în fiecare nod: data la care s-a jucat meciul și scorul; pe baza scorului, în nodul părinte va fi scris numele echipei câștigătoare. După fiecare execuție a programului, arborele trebuie salvat într-un fișier, de unde trebuie readus la următoarea execuție.

2.8.5. Arborele binar de căutare

2.8.5.1. Definiția arborelui binar de căutare

Terminologie:

- Se numește **cheie** un câmp din informația utilă a nodului care poate fi folosit pentru a identifica unic nodurile arborelui.

Se numește **arbore binar de căutare** un arbore binar care are proprietatea că, pentru fiecare nod, cheia din **succesorul stâng este mai mică decât cheia din nod, iar cheia din successorul drept este mai mare decât cheia din nod.**

Exemplu – Arborele binar A₁₄ din figura 79.

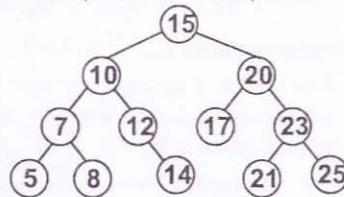


Fig. 79

A₁₄

Caracteristici:

- Pentru orice nod, subarborele stâng conține noduri cu valori mai mici ale cheii, iar subarborele drept conține noduri cu valori mai mari ale cheii.
 → Într-un arbore binar de căutare **nu există două noduri cu aceeași valoare a cheii.**



Pentru multimea de chei {2, 5, 7, 10, 12, 15, 23} desenați arborii binari de căutare cu înălțimea 2, 3, 4, 5 și 6.

2.8.5.2. Algoritmi pentru prelucrarea unui arbore binar de căutare

Pentru prelucrarea unui arbore binar de căutare se pot folosi următorii **algoritmi**:

- Algoritmul pentru **creare**,
- Algoritmii pentru **parcursere**,
- Algoritmii pentru **actualizare**.

Algoritmul pentru crearea unui arbore binar de căutare

Crearea unui arbore binar de căutare se face prin adăugarea fiecărui nod ca nod terminal, în poziția corespunzătoare, astfel încât arborele să nu-și piardă proprietatea din definiție. Căutarea nodului la care se va adăuga ca succesor nodul terminal și ce fel de succesor va fi (stâng sau drept) se face cu ajutorul unui pointer care va indica nodul curent care se analizează. Nodul de la care se pornește este rădăcina arborelui. Cu ajutorul acestui pointer se avansează pe nivelurile arborelui prin succesorul stâng sau succesorul drept al nodului curent, în funcție de rezultatul comparației dintre valoarea citită pentru cheie și valoarea cheii din nodul curent. Avansarea continuă până când succesorul nodului curent la care a ajuns pointerul este arborele vid. Algoritmul pentru crearea arborelui binar de căutare este:

PAS1. Se inițializează arborele cu arborele vid, atribuind rădăcinii adresa NULL.

PAS2. Se citește informația utilă, cu valoarea **v** pentru cheie.

PAS3. Cât timp există informație utilă **execută**:

PAS4. Se inițializează nodul curent cu nodul rădăcină.

PAS5. Dacă nodul curent nu este un arbore vid, **atunci** se trece la Pasul 9.

PAS6. Se creează un nod prin alocarea unei zone de memorie.

PAS7. Se atribuie, câmpului cu informație din nod, informația citită.

PAS8. Se atribuie, succesorilor nodului, arborele vid. Se trece la Pasul 11.

PAS9. Dacă **v** (cheia din informație), este mai mică decât cheia nodului curent, **atunci** nodul curent devine succesorul stâng și se revine la Pasul 5.

PAS10. Dacă **v** (cheia din informație) este mai mare decât cheia nodului curent, **atunci** nodul curent devine succesorul drept și se revine la Pasul 5; **altfel**, se afișează mesajul "Cheia există deja".

PAS11. Se citește informația utilă cu valoarea **v** pentru cheie.

Implementarea subprogramului prin care se adaugă un nod la arbore s-a făcut recursiv.

Observație. Dacă se prelucreză o mulțime de valori numerice care nu sunt diferite între ele, se poate folosi arborele binar de căutare, adăugând la structura nodului un câmp în care se numără frecvența de apariție a numărului în sirul de valori (**frecv**):

```
struct nod {int nr,frecv;
            nod *s,*d;};
```

În acest mod, este respectată condiția de cheie unică impusă de definiția arborelui binar de căutare, în nodurile lui fiind memorate numai valorile unice ale cheii. Programul de creare a unui arbore binar de căutare, în cele două variante, este:

Valoarea pentru cheie este unică

```
void creare(nod *&r, int n)
{if (r!=NULL)
  if (n<r->nr) creare(r->s,n);
   else if (n>r->nr) creare(r->d,n);
   else
     cout<<"Numarul există" << endl;
   else {r = new nod;
          r->nr=n;
          r->s=NULL;
          r->d=NULL;}}
void main()
{int n; r=NULL;
cout<<"Numar: "; cin>>n;
```

Valoarea pentru cheie nu este unică

```
void creare(nod *&r, int n)
{if (r!=NULL)
  if (n<r->nr) creare(r->s,n);
   else if (n>r->nr) creare(r->d,n);
   else r->frecv++;
   else {r = new nod;
          r->nr=n;
          r->frecv=1;
          r->s=NULL;
          r->d=NULL;}}
void main()
{int n; r=NULL;
cout<<"Numar: "; cin>>n;
```

```
while (n!=0)
{creare(r,n);
 cout<<"Numar: "; cin>>n;
...}
```

```
while (n!=0)
{creare(r,n);
 cout<<"Numar: "; cin>>n;
...}
```

Observație. Înălțimea arborelui binar de căutare depinde de ordinea în care se introduc valorile cheii.

Temă

- Pentru fiecare dintre arborii binari de căutare desenați la tema anterioară, precizați în ce ordine au fost citite cheile, la crearea arborelui.
- Desenați arborii binari de căutare care se creează atunci când introduceți pentru cheie, în ordine, următoarele valori:
 - 3, 6, 2, 7, 8, 1, 5, 4, 0;
 - 7, 3, 1, 2, 8, 5, 4, 6, 0;
 - 1, 2, 3, 4, 5, 6, 7, 8, 0.

Algoritmi pentru parcurgerea unui arbore binar de căutare

Pentru parcurgerea arborilor binari de căutare se folosesc algoritmii de parcugere a unui arbore binar.

Observații

- Prin parcugerea arborelui cu **algoritmul SRD** cheile sunt afișate în ordine **crescătoare**.
- Prin parcugerea arborelui cu **algoritmul DRS** cheile sunt afișate în ordine **descrescătoare**.
- Cheia cu **valoarea maximă** se găsește în nodul cel mai din stânga, iar căutarea sa se face parcugând arborele numai pe legătura cu succesorul stâng, pornind din rădăcină.
- Cheia cu **valoarea minimă** se găsește în nodul cel mai din dreapta, iar căutarea sa se face parcugând arborele numai pe legătura cu succesorul drept, pornind din rădăcină.

Valoarea minimă

Implementare recursivă

```
int min(nod *r)
{if (r->s!=NULL)
    return min(r->s);
 else return r->nr;}
void main()
{ ...
cout<<endl<<"Minim: "<<min(r); ...}
```

```
int max(nod *r)
{if (r->d!=NULL)
    return max(r->d);
 else return r->nr;}
void main()
{ ...
cout<<endl<<"Maxim: "<<max(r); ...}
```

Implementare iterativă

```
int min(nod *r)
{while (r->s!=NULL)
    r=r->s;
 return r->nr;}
void main()
{ ...
cout<<endl<<"Minim: "<<min(r); ...}
```

```
int max(nod *r)
{while (r->d!=NULL)
    r=r->d;
 return r->nr;}
void main()
{ ...
cout<<endl<<"Maxim: "<<max(r); ...}
```

- Deoarece între nodurile arborelui binar de căutare este stabilită o relație de ordine, **căutarea** unei anumite chei se face rapid prin mecanismul căutării binare (se caută în una din cele două submulțimi de valori – subarborele stâng sau subarborele drept). Căutarea nodului se face cu ajutorul unui pointer, care pornește din nodul rădăcină și care va indica nodul curent care se analizează. Pointerul va avansa prin succesorul stâng sau succesorul drept al nodului curent, în funcție de rezultatul comparației dintre

valoarea citită pentru cheie și valoarea cheii din nodul curent. Avansarea continuă până când se găsește un nod care conține cheia căutată sau până se ajunge la un arbore vid (cazul în care nu există cheia căutată în arbore) – figura 80.

PAS1. Se citește cheia k care se caută.

PAS2. Se inițializează pointerul cu adresa rădăcinii.

PAS3. Cât timp nu s-a găsit cheia căutată și nu s-a ajuns la arbore vid **execută**:

PAS4. Dacă nodul curent are cheia mai mică decât k , atunci pointerul avansează la subarborele stâng; altfel, pointerul avansează la subarborele drept.

Implementare recursivă

```
nod *cauta(nod *r,int k)
{if (r=NULL || r->nr==k) return r;
 else
 if(r->nr>k) return cauta(r->s,k);
 else return cauta(r->d,k);}
void main()
{int k; ...
 cout<<"Cheia cautata: "; cin>>k;
 if (cauta(r,k)!=NULL)
 cout<<"S-a gasit";
 else cout<<"Nu s-a gasit"; ...}
```

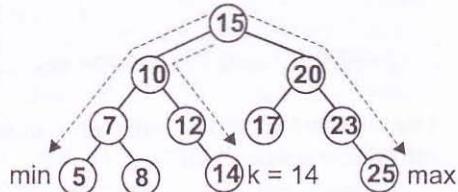


Fig. 80

Implementare iterativă

```
nod *cauta(nod *r,int k)
{while (r!=NULL && r->nr!=k)
 if(r->nr>k) r=r->s;
 else r=r->d;
 return r;}
void main()
{int k; ...
 cout<<"Cheia cautata: "; cin>>k;
 if (cauta(r,k)!=NULL)
 cout<<"S-a gasit";
 else cout<<"Nu s-a gasit"; ...}
```

Algoritmi pentru actualizarea unui arbore binar de căutare

În arborii binari de căutare se pot executa următoarele operații de actualizare:

→ inserarea unui nod;

→ ștergerea unui nod.

Atenție

În urma acestor operații de actualizare, arborele trebuie să-și păstreze calitatea de arbore binar de căutare.

Algoritmul pentru inserarea unui nod

Căutarea poziției în care se inserează nodul se face cu ajutorul unui pointer, care pornește din nodul rădăcină și care va indica nodul curent care se analizează. Pointerul va avansa prin succesorul stâng sau succesorul drept al nodului curent, în funcție de rezultatul comparației dintre valoarea citită pentru cheia care se inserează și valoarea cheii din nodul curent. Avansarea continuă până când se găsește un nod care conține o cheie egală cu cheia care se inserează (cazul în care nodul nu se poate insera, deoarece mai există o cheie cu aceeași valoare în arbore) sau până când, în nodul curent, succesorul pe care avansează pointerul este un arbore vid. Dacă s-a ajuns la arborele vid, înseamnă că valoarea cheii care se inserează nu mai există în arbore – și cheia se adaugă ca succesor al nodului curent, drept sau stâng, în funcție de valoarea cheii care se inserează și de valoarea cheii din nodul curent. Variabila r este pointerul către nodul care se inserează. În varianta iterativă, variabila c este pointerul către nodul curent, iar variabila logică x se folosește pentru a ști dacă în arbore s-a găsit cheia care se inserează (este inițializată cu valoarea 1, presupunându-se că nu mai există cheia în arbore).

PAS1. Se citește cheia k a nodului care se inserează.

PAS2. Se initializează pointerul r cu adresa rădăcinii.

PAS3. Cât timp nu s-a ajuns la arbore vid și nu s-a găsit cheia care se inserează **execută**:

PAS4. Dacă nodul curent are cheia egală cu k , **atunci** se afișează mesajul "Cheia există", se atribuie lui x valoarea 0 și se trece la Pasul 6.

PAS5. Dacă nodul curent are cheia mai mică decât k , **atunci** pointerul avansează la subarborele stâng; **altfel**, pointerul avansează la subarborele drept și se revine la Pasul 3.

PAS6. Dacă nu s-a găsit în arbore cheia care se inserează ($x <> 0$), **atunci**:

PAS7. Se creează un nod prin alocarea unei zone de memorie.

PAS8. Se atribuie, câmpului cu informație din nod, informația care se inserează.

PAS9. Se atribuie, succesorilor nodului, arborele vid.

PAS10. Dacă nodul curent c are cheia mai mică decât k , **atunci** nodul care se inserează este succesorul drept; **altfel**, nodul care se inserează este succesorul stâng.

Valoarea pentru cheie este unică

Implementare recursivă

```
void inserare(nod *&r, int k)
{
    if (r!=NULL)
        if (r->nr==k)
            cout<<"Cheia există" << endl;
        else if (r->nr>k)
            inserare(r->s, k);
        else inserare(r->d, k);
    else {r=new nod; r->nr=k;
          r->s=NULL; r->d=NULL;}
}
void main()
{int k; ...
cout<<"Cheia care se insereaza: ";
cin>>k; inserare(r, k); ...}
```

Valoarea pentru cheie nu este unică

```
void inserare(nod *&r, int k)
{
    if (r!=NULL)
        if (r->nr==k) r->freqv++;
        else if (r->nr>k)
            inserare(r->s, k);
        else inserare(r->d, k);
    else {r=new nod; r->nr=k;
          r->s=NULL; r->d=NULL;}
}
void main()
{int k; ...
cout<<"Cheia care se insereaza: ";
cin>>k; inserare(r, k); ...}
```

Implementare iterativă

```
void inserare(nod *&r, int k)
{
    int x=1; nod *c;
    while (r!=NULL && x)
        if (r->nr==k)
            {cout<<"Cheia există" << endl;
             x=0;};
        else
            if (r->nr>k) {c=r; r=r->s;}
            else {c=r; r=r->d;}
    if (x)
        {r=new nod; r->nr=k;
         r->s=NULL; r->d=NULL;
         if (c->nr>k) c->s=r;
         else c->d=r;}
}
void main()
{int k; ...
cout<<"Cheia care se insereaza: ";
cin>>k; inserare(r, k); ...}
```

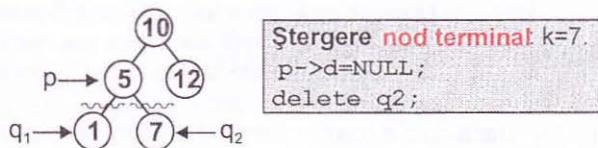
```
void inserare(nod *&r, int k)
{
    int x=1; nod *c;
    while (r!=NULL && x)
        if (r->nr==k)
            {r->freqv++; x=0;};
        else
            if (r->nr>k) {c=r; r=r->s;}
            else {c=r; r=r->d;}
    if (x)
        {r=new nod; r->nr=k;
         r->s=NULL; r->d=NULL;
         if (p->nr>k) c->s=r;
         else c->d=r;};
}
void main()
{int k; ...
cout<<"Cheia care se insereaza: ";
cin>>k; inserare(r, k); ...}
```

Algoritmul pentru ștergerea unui nod

La ștergerea unui nod din arborele binar pot să apară trei cazuri:

- Nodul care se șterge nu are fiu.** În nodul părinte, adresa lui este înlocuită cu adresa arborelui vid – figura 81.

Ștergere nod terminal k=1.
 $p \rightarrow s = \text{NULL};$
 $\text{delete } q_1;$



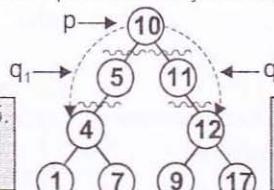
Ștergere nod terminat k=7.
 $p \rightarrow d = \text{NULL};$
 $\text{delete } q_2;$

Fig. 81

- Nodul care se șterge are un fiu.** În nodul părinte, adresa lui este înlocuită cu adresa fiului (se stabilește o legătură între părintele lui și fiul lui) – figura 82.

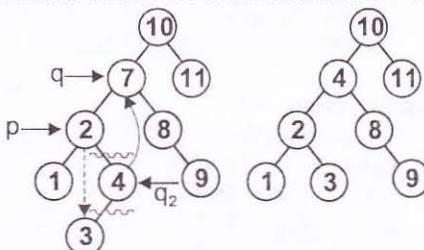
Fig. 82

Ștergere nod cu fiu stâng k=5.
 $p \rightarrow s = p \rightarrow s \rightarrow s;$
 $\text{delete } q_1;$



Ștergere nod cu fiu drept k=11.
 $p \rightarrow d = p \rightarrow d \rightarrow d;$
 $\text{delete } q_2;$

- Nodul care se șterge are doi fiu.** Se caută, în subarborele drept al nodului care trebuie șters, primul nod care nu are succesor drept. Acest nod are cea mai mare cheie din subarborele drept și, implicit, cea mai mare cheie din subarborele subordonat nodului care se șterge. Informația utilă din acest nod este copiată în nodul care trebuie șters, după care este șters nodul găsit ca nod cu un singur fiu – succesorul stâng (cazul 2). Prin copierea informației din nodul cu cheia cea mai mare, arborele își păstrează proprietatea de arbore binar de căutare – figura 83.



Ștergere nod cu doi fiu k=7.
 $q \rightarrow \text{info} = r \rightarrow \text{info};$
 $p \rightarrow s = p \rightarrow s \rightarrow d;$
 $\text{delete } r;$

Fig. 83

PAS1. Se citește cheia k a nodului care se șterge.

PAS2. Se initializează pointerul r cu adresa rădăcinii.

PAS3. Cât timp nu s-a șters nodul și nu s-a ajuns la arborele vid **execuță**:

PAS4. Dacă nodul curent are cheia egală cu k, atunci se trece la Pasul 5, pentru a se analiza ce tip de nod este; altfel, se trece la Pasul 9.

PAS5. Dacă nodul curent este nod terminal, atunci, în nodul părinte, adresa lui este înlocuită cu arborele vid, se eliberează zona de memorie ocupată de nodul curent – și se revine la Pasul 3.

PAS6. Dacă nodul curent are numai succesor drept, atunci, în nodul părinte, adresa lui este înlocuită cu adresa succesorului său drept, se eliberează zona de memorie ocupată de nodul curent – și se revine la Pasul 3.

- PAS7.** Dacă nodul curent are numai succesor stâng, atunci, în nodul părinte, adresa lui este înlocuită cu adresa succesorului său stâng, se eliberează zona de memorie ocupată de nodul curent – și se revine la Pasul 3.
- PAS8.** Se caută, în subarborele drept al nodului curent, primul nod care nu are succesor drept, se copiază informația utilă din acest nod în nodul curent, se eliberează zona de memorie ocupată de nodul găsit – și se revine la Pasul 3.
- PAS9.** Dacă nodul curent are cheia mai mică decât k , atunci se caută nodul care trebuie șters în succesorul drept; altfel, se caută nodul care trebuie șters în succesorul stâng. Se revine la Pasul 3.

PAS10. Dacă s-a ajuns la arborele vid, atunci se afișează mesajul "Cheia nu există",

Observație. Transmiterea adresei de la nodul curent (care se șterge) la părintele său, se face prin transmiterea prin referință a parametrului cu adresa nodului.

```
void sterge_nod(nod *&r, nod *&c)
/*se caută în subarborele drept al nodului curent (r) primul nod care
nu are succesor drept (c) */
{nod *p;
 if (c->d!=NULL) sterge_nod(r,c->d);
 else {r->nr=c->nr; //se copiază informația utilă din nodul
       //găsit în nodul curent
       p=c->s; delete c; c=p; }
//se șterge nodul găsit ca nod care are numai succesor drept
void stergere(nod *&r,int k)
{nod *c;
 if (r!=NULL)
 if (r->nr==k) //dacă s-a găsit nodul care trebuie șters
 if (r->s==NULL && r->d==NULL) //dacă este nod terminal
 {delete r; r=NULL;}
 else
 if (r->s==NULL) //dacă este nod care are numai succesor drept
 {c=r->d; delete r; r=c;}
 else if (r->d==NULL) //dacă este nod care are numai succesor stâng
 {c=r->s; delete r; r=c;}
 else sterge_nod(r,r->s); //dacă are ambii succesiuni
 else //dacă nu s-a găsit nodul care trebuie șters
 if (r->nr<k) stergere(r->d,k); //caută în succesorul drept
 else stergere(r->s,k); //caută în succesorul stâng
 else cout<<"Cheia nu există";}
void main()
{int k; ... cout<<"Cheia care se sterge: "; cin>>k; stergere(r,k); ...}
```

Complexitatea algoritmilor de prelucrare a arborilor binari de căutare

Căutarea unei informații identificată printr-o cheie cu valoarea k începe de la nodul rădăcină și se termină, în cel mai rău caz, într-unul dintre nodurile terminale ale arborelui. Găsirea nodului presupune executarea operației de comparare dintre cheia cu valoarea k și cheia nodului curent. Timpul mediu de execuție al algoritmului se calculează la fel ca la algoritmul de căutare secvențială într-o structură liniară de n chei (reprezentate sub formă de vector sau listă înlănțuită), unde:

$$T_{\text{med}} = \frac{n+1}{2}$$

În cazul arborelui binar de căutare, dimensiunea datelor nu este n , ci numărul de niveluri ale arborelui de căutare (sau înălțimea lui). Înălțimea fiind cu 1 mai mică decât numărul de niveluri parcuse (se consideră și nivelul rădăcinii), în cazul arborelui binar de căutare:

$$T_{med} = \frac{h + 2}{2}$$

Ordinul de complexitate al algoritmului este $O(h)$. Rezultă că timpul consumat pentru operațiile de prelucrare a arborilor binari de căutare este direct proporțional cu înălțimea arborelui h .

Propoziția 20

Înălțimea unui arbore binar cu n noduri poate lua valori între $\log_2(n)-1$ și $n-1$.

Demonstrație. Notăm cu h înălțimea arborelui binar cu n noduri. Arboarele binar are cea mai mare înălțime (h_{max}) atunci când este degenerat (nodurile sunt distribuite câte unul pe fiecare nivel – cu excepția nodului terminal, ordinul fiecărui nod este 1). Rezultă că $h_{max}=n-1$ și $h \leq n-1$. Arboarele binar are cea mai mică înălțime (h_{min}) atunci când este complet (toate nodurile terminale se găsesc pe același nivel – și ordinul fiecărui nod, cu excepția nodurilor terminale, este 2). În acest caz, pe fiecare nivel i vor fi 2^i noduri, iar numărul n de noduri va fi egal cu $2^{h_{min}+1}-1$. Rezultă că $n+1=2^{h_{min}+1}$ și $h_{min}+1=\log_2(n+1)$. Din ultima egalitate rezultă că $h \geq \log_2(n)-1$.

Eficiența algoritmului de prelucrare a arborilor binari depinde de modul în care a fost creat arborele. Dacă el are înălțimea maximă (arboarele binar degenerat, în care fiecare nod nu are decât un succesor), ordinul de complexitate al algoritmului va fi $O(n)$ și este egal cu cel al algoritmului de căutare secvențială într-o structură de date liniară. Dacă el are înălțimea minimă (arborele complet sau aproape complet), ordinul de complexitate al algoritmului va fi $O(\log_2 n)$.

Temă



1. Comparați, din punct de vedere al eficienței, algoritmii recursivi și iterativi pentru determinarea minimului, respectiv a maximului, într-un arbore binar de căutare. Care dintre variante este mai eficientă?
2. Pentru arboarele binar A_{14} din figura 79, câte operații de deplasare în structura de date se execută pentru a determina minimul? Dacă valorile din arbore ar fi fost memorate într-o structură liniară, câte operații de deplasare s-ar fi executat pentru a determina minimul? Care dintre structurile de date este mai eficientă?
3. Scrieți o funcție care să creeze un arbore binar de căutare perfect echilibrat.
4. Scrieți o funcție care să verifice dacă un arbore binar este un arbore binar de căutare.
5. În nodurile unui arbore binar sunt memorate numărătorul și numitorul unor fracții distințe. Să se simplifice fracțiile din arbore, iar dacă în urma simplificării rezultă un număr întreg, să se eliminate din arbore.
6. În nodurile unui arbore binar sunt memorate numere întregi. Se citește de la tastatură un număr întreg x . Să se caute în arbore nodul cheia x și dacă se găsește, să se afișeze fratele său – precizându-se dacă este fratele stâng sau fratele drept. Să se afișeze un mesaj de informare dacă nu se găsește cheia x în arbore sau dacă nodul nu are frate.

2.8.5.3. Aplicații practice

1. Un text conține cuvinte care se pot repeta. Textul este memorat într-un fișier text. Scrieți un program care să citească textul din fișier și care să afișeze cuvintele din text, ordonate alfabetic, și frecvența lor de apariție în text.
2. Realizați o agendă de telefon cu ajutorul unui arbore binar de căutare. În fiecare nod se vor păstra următoarele informații: numele și prenumele persoanei, numărul (numerele de telefon) și adresa de e-mail. Scrieți un program care să permită adăugarea de noi persoane în agendă, eliminarea unor persoane și căutarea unei persoane pentru a afla numărul de telefon și adresa de e-mail.

2.8.6. Ansamblul Heap

2.8.6.1. Definiția ansamblului Heap

Se numește **ansamblu Heap** un arbore binar care îndeplinește următoarele condiții:

- (1) este un **arbore aproape complet**;
- (2) în orice pereche de noduri **tată-fiu** cheile sunt într-o **relație de ordine prestabilită**.

Exemplu. În figura 84 este prezentat ansamblul Heap A₁₅ în nodurile căruia sunt memorate numere naturale, relația de ordine fiind: dacă în nodul tată este un număr impar, în nodul fiu nu poate fi decât tot un număr impar. Numerele scrise în exteriorul nodurilor, reprezintă indicii pentru numerotarea nodurilor.

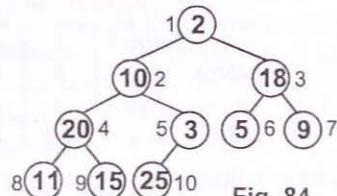


Fig. 84

Terminologie:

- Ansamblul Heap se mai numește și **arbore de selecție** sau **arbore parțial ordonat**.
- Un ansamblu **Heap maxim** este un ansamblu Heap în care cheia părintelui este mai mare sau egală cu cheia fiului (arborele binar A₁₆ din figura 85).
- Un ansamblu **Heap minim** este un ansamblu Heap în care cheia părintelui este mai mică sau egală cu cheia fiului (arborele binar A₁₇ din figura 86).

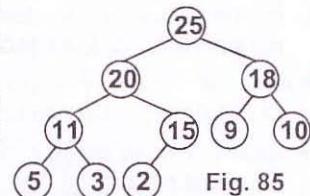


Fig. 85

Caracteristici:

- Într-un ansamblu Heap cu n noduri, înălțimea arborelui h este egală cu $\lceil \log_2 n \rceil - 1$.
- Într-un **ansamblu Heap** (de exemplu, într-un ansamblu **Heap minim** sau **Heap maxim**) pot exista chei cu aceeași valoare.
- Într-un ansamblu **Heap minim** cheia din orice nod este mai mică sau egală cu cheile tuturor nodurilor din cei doi subarbore ai săi, iar în **nodul rădăcină** se găsește **cheia cea mai mică** din arbore.
- Într-un ansamblu **Heap maxim** cheia din orice nod este mai mare sau egală cu cheile tuturor nodurilor din cei doi subarbore ai săi, iar în **nodul rădăcină** se găsește **cheia cea mai mare** din arbore.
- Un ansamblu Heap poate fi folosit pentru a implementa o **coadă de priorități** (de aici îi vine și denumirea de **arbore de selecție**), deoarece – prin extragerea cheilor prin nodul rădăcină – se extrag datele în conformitate cu relația de ordine prestabilită. Pentru ansamblul Heap A₁₅ din figura 85, ordinea de extragere a datelor este: mai întâi numerele pare și apoi numerele impare. În ansamblul Heap A₁₆ din figura 86 extragerea nodurilor se face în ordinea descrescătoare a etichetelor, iar în ansamblul Heap A₁₇ din figura 86 extragerea nodurilor se face în ordinea crescătoare a etichetelor.

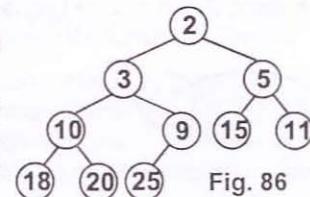


Fig. 86

Proprietate: Ansamblul Heap are o proprietate foarte importantă (pe care o moștenește de la arborii binari compleți): dacă nodurile arborelui se numerotează cu indicii, în ordinea parcurgerii în lățime (ca în figura 84), între indicii părintelui, al subarborelui stâng și al subarborelui drept – există următoarele relații:

- **Părintele** nodului cu indicele i are indicele $\left\lceil \frac{i}{2} \right\rceil$.

→ **Fiul stâng** al părintelui cu indicele i are indicele $2 \times i$.

→ **Fiul drept** al părintelui cu indicele i are indicele $2 \times i + 1$.

Această proprietate a arborilor compleți și aproape compleți ne permite să implementăm static ansamblul Heap, cu ajutorul unui vector v.

→ Rădăcina arborelui are eticheta $v[1]$.

→ **Fiul stâng** al nodului cu eticheta $v[i]$ este nodul cu eticheta $v[2 \times i]$.

→ **Fiul drept** al nodului cu eticheta $v[i]$ este nodul cu eticheta $v[2 \times i + 1]$.

	Indice	1	2	3	4	5	6	7	8	9	10
vectorul	A_{16}	2	10	18	20	3	5	9	11	15	25
v	A_{17}	2	3	5	10	9	15	11	18	20	25
pentru	A_{18}	25	20	18	11	15	9	10	5	3	2

Observații:

1. Într-un vector care implementează un ansamblu **Heap Maxim**: $v[i] \geq v[2 \times i]$ și $v[i] \geq v[2 \times i + 1]$ pentru $\forall i \in \{1, 2, 3, \dots, [n/2]\}$ și dacă $2 \times i \leq n$, respectiv $2 \times i + 1 \leq n$.
2. Într-un vector care implementează un ansamblu **Heap Minim**: $v[i] \leq v[2 \times i]$ și $v[i] \leq v[2 \times i + 1]$ pentru $\forall i \in \{1, 2, 3, \dots, [n/2]\}$ și dacă $2 \times i \leq n$, respectiv $2 \times i + 1 \leq n$.

Invers, **un vector v – poate fi reprezentat ca un arbore binar aproape complet**. Pentru un indice $i \in \{1, 2, 3, \dots, [n/2]\}$:

- dacă $2 \times i > n$, se poate construi un subansamblu Heap care este format numai din nodul părinte $v[i]$ (figura 87);
- dacă $2 \times i = n$, se poate construi un subansamblu Heap care este format din nodul părinte $v[i]$ și fiul stâng $v[2 \times i]$ (figura 88);
- dacă $2 \times i < n$ și $2 \times i + 1 \leq n$, se poate construi un subansamblu Heap care este format din nodul părinte $v[i]$, fiul stâng $v[2 \times i]$ și fiul drept $v[2 \times i + 1]$ (figura 89).

V [i]

Fig. 87

V [i]

Fig. 88

V [i]

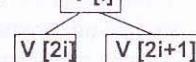


Fig. 88



Tema Pentru multimea de chei {2, 5, 7, 10, 12, 15, 23, 25, 28} scrieți vectorul corespunzător ansamblului **Heap maxim** și vectorul corespunzător ansamblului **Heap minim**. Desenați ansamblurile Heap pornind de la vectori.

V [2i] V [2i+1]

Fig. 89

2.8.6.2. Algoritmi pentru prelucrarea unui ansamblu Heap

Pentru prelucrarea unui ansamblu Heap se pot folosi următorii algoritmi:

- Algoritmul pentru **crearea** ansamblului Heap,
- Algoritmul pentru **inserarea** unui nod,
- Algoritmul pentru **extragerea** unui nod.



În urma acestor operații de prelucrare, arboarele trebuie să-și păstreze calitatea de ansamblu Heap.

În algoritmii pentru prelucrarea unui ansamblu Heap se va folosi implementarea statică a ansamblului cu ajutorul vectorului. Dacă lungimea fizică a vectorului este NMAX, numărul de noduri n ale ansamblului Heap nu trebuie să fie mai mare decât NMAX-1. Deoarece vectorul Heap se construiește pe baza mecanismului de legătură părinte-fii între indici, etichetele ansamblului Heap se vor memora în vector începând cu indicele 1.

Algoritmul pentru crearea unui ansamblu Heap

Crearea unui ansamblu Heap presupune adăugarea elementelor în vector, pornind de la primul element, astfel încât această reprezentare să respecte proprietățile unui ansamblu Heap:

- (1) Să fie un **arbore aproape complet**. Deoarece în arbore nodul trebuie adăugat pe ultimul nivel, imediat la dreapta nodului existent, înseamnă că în vector se va adăuga noua cheie imediat după ultimul element.
- (2) În orice pereche de noduri **tată-fiu**, cheile trebuie să fie în **relația de ordine** prestatibilită. Dacă în arbore se adaugă un nod în poziția precizată, este posibil ca el să nu fie cu părintele său în relația de ordine prestatabilită. Soluția este de a propaga informația din acest nod către rădăcină, până când este îndeplinită relația de ordine între nodul în care a ajuns informația și părintele său.

În algoritmul pentru crearea ansamblului Heap se folosește variabila **j** pentru lungimea logică a vectorului (reprazintă numărul de elemente care au fost adăugate în vector) și variabila **i** pentru indicele nodului curent. Algoritmul următor se poate folosi pentru crearea unui **Heap Maxim**:

PAS1. Se citește prima cheie și se memorează în rădăcina arborelui (elementul **v[1]**).

PAS2. Pentru nodul **j** de la al doilea nod, până la ultimul nod din arbore, **execută**:

PAS3. Se scrie cheia în elementul cu indicele **j**.

PAS4. Se inițializează indicele elementului curent **i** cu lungimea logică a vectorului **j**.

PAS5. Cât timp nodul curent este diferit de nodul rădăcină **execută**:

PAS6. Dacă cheia memorată în nodul curent (elementul cu indicele **i**) este mai mare decât cheia memorată în nodul părinte al nodului curent (elementul cu indicele **i/2**), **atunci** se interschimbă cheia din nodul curent cu cheia din nodul părinte, iar nodul părinte devine nod curent; **altfel**, rădăcina devine nod curent.

```

fstream f("heap.txt",ios::in); //cheile se citesc din fisier
int const NMAX=100;
int v[NMAX],n;
void creare_heap()
{int i,j,aux; f>>n;
 if (n<=NMAX-1)
 {f>>v[1];
  for (j=2;j<=n;j++)
  {f>>v[j]; i=j;
   while (i>1)
    if (v[i]>=v[i/2]) {aux=v[i],v[i]=v[i/2], v[i/2]=aux; i=i/2;}
    else i=1;}}
 else cout<<"Nu se poate crea - arborele are mai multe noduri
 decat lungimea fizica a vectorului"<<endl;}
void afisare() {for (int i=1;i<=n;i++) cout<<v[i]<<" ";}
void main() {creare_heap(); afisare();}

```

Algoritmul pentru inserarea unui nod în ansamblul Heap

Prin adăugarea unui nou nod, lungimea logică a vectorului crește cu 1 și trebuie să se verifice să nu depășească lungimea fizică a vectorului. Prin inserarea unui nod, arborele trebuie să-și păstreze calitatea de ansamblu Heap, adică să aibă proprietățile:

- (1) Să fie un **arbore aproape complet**. În arbore, nodul se va adăuga pe ultimul nivel, imediat la dreapta nodului existent. Dacă ultimul nivel este complet, nodul se va adăuga

pe următorul nivel, ca succesor stâng al nodului din extremitatea stângă de pe nivelul anterior. Înseamnă că în vector se va adăuga noua cheie imediat după ultimul element.

- (2) În orice pereche de noduri **tată-fiu** cheile trebuie să fie în **relația de ordine** prestaabilită: Dacă în arbore se adaugă un nod în poziția precizată, este posibil ca el să nu fie cu părintele său în relația de ordine prestabilită – și informația trebuie să se propage din acest nod către rădăcină, la fel ca la algoritmul de creare. Înseamnă că, în vector, ultimul element va fi considerat element curent – și cheia memorată în ele se va compara cu cheia memorată în părinte. Dacă nu respectă relația de ordine, se interschimbă cheia nodului curent cu cheia părintelui, iar părintele devine nod curent. Propagarea cheii continuă până când între elementul curent și părintele său este respectată relația de ordine, sau până când rădăcina ajunge să fie element curent.

```
...
void adauga() //pentru ansamblul Heap Maxim
{int x,i;
 if (n==NMAX-1) cout<<"Nu se mai poate adauga nodul "<<endl;
 else
 {cout<<"cheia= "; cin>>x; n++; v[n]=x; i=n;
  while (i>1)
   if (v[i]>=v[i/2]) {x=v[i]; v[i]=v[i/2]; v[i/2]=x; i=i/2;}
   else i=1;}}
void main() {creare(); ... adauga(); ... }
```

Algoritmul pentru extragerea unui nod din ansamblul Heap

Ansamblul Heap, fiind o coadă de priorități, prima informație care se prelucrează este cea din nodul rădăcină. Extragerea unui nod din ansamblul Heap înseamnă de fapt prelucrarea informației din nodul rădăcină și eliminarea acestui nod din arbore. Pentru ca arborele să-și păstreze calitatea de ansamblu Heap, după eliminarea nodului rădăcină trebuie să aibă proprietățile:

- (1) Să fie un **arbore aproape complet**. Înseamnă că trebuie să dispară din arbore nodul cel mai din dreapta de pe ultimul nivel. Dacă din arbore dispăr nodul din dreapta de pe ultimul nivel, nu trebuie să dispară din arbore și informația din acest nod – din arbore trebuie să dispară informația din nodul rădăcină. De aceea, în nodul rădăcină este adusă informația din ultimul nod. În vector, cheia din ultimul element va fi salvată în primul element, după care, este eliminat ultimul element prin decrementarea lungimii logice a vectorului.
- (2) În orice pereche de noduri **tată-fiu** cheile trebuie să fie în **relația de ordine** prestaabilită. Dacă în nodul rădăcină este adusă informația din ultimul nod, este foarte probabil să nu mai fie respectată relația de ordine între nodul rădăcină și fiili săi. Solutia este de a propaga informația adusă în rădăcină, către nivelurile inferioare, până când este îndeplinită relația de ordine între nodul în care a ajuns informația și fiili săi.

În algoritmul pentru eliminarea nodului din ansamblul Heap se folosește variabila **i** pentru indicele nodului curent, iar variabila **j** – pentru indicele nodului fiu al nodului curent.

- PAS1.** Se prelucrează cheia din rădăcină (elementul **v[1]**).
- PAS2.** Se copiază în rădăcină cheia din ultimul nod (elementul **v[n]**) și se micșorează cu 1 lungimea logică a vectorului.
- PAS3.** Se initializează nodul curent cu nodul rădăcină (indicele **i** are valoarea 1).
- PAS4.** Cât timp nodul curent face parte din arbore (elementul cu indicele **i** face parte din vector) **execută**:

- PAS5.** Dacă nodul curent are succesor stâng, atunci fiul său va fi succesorul stâng (indicele j este egal cu 2^*i); altfel, nodul curent este scos în afara arborelui, atribuind indicelui i o valoare mai mare decât lungimea logică a vectorului – și se revine la Pasul 4.
- PAS6.** Dacă nodul curent are și succesor drept, și cheia din succesorul stâng este mai mică decât cheia din succesorul drept, atunci fiul nodului curent va fi succesorul drept (indicele j este egal cu 2^*i+1).
- PAS7.** Dacă cheia memorată în nodul curent (elementul cu indicele i) este mai mică decât cheia memorată în nodul fiu al nodului curent (elementul cu indicele j), atunci se interschimbă cheia din nodul curent cu cheia din nodul fiu, iar nodul fiu devine nod curent; altfel, nodul curent este scos în afara arborelui. Se revine la Pasul 4.

```
...
int elimina()      //pentru ansamblul Heap Maxim
{int i,j,x=v[1],aux; v[1]=v[n]; n--; i=1;
while (i<=n)
{if (2*i<=n)
{j=2*i;
 if (j+1<=n && v[j+1]>=v[j]) j++;
 if (v[i]<=v[j]){aux=v[i]; v[i]=v[j]; v[j]=aux; i=j;}
 else i=n+1;}
else i=n+1;}
return x;}
void main()
{creare();...cout<<"Cheia cea mai mare este "<<elimina()<<endl;...}
```

Observație. Dacă se extrag din arbore toate nodurile și se afișează cheile nodurilor extrase, cheile vor fi afișate în relația de ordine prestabilită pentru ansamblul Heap.

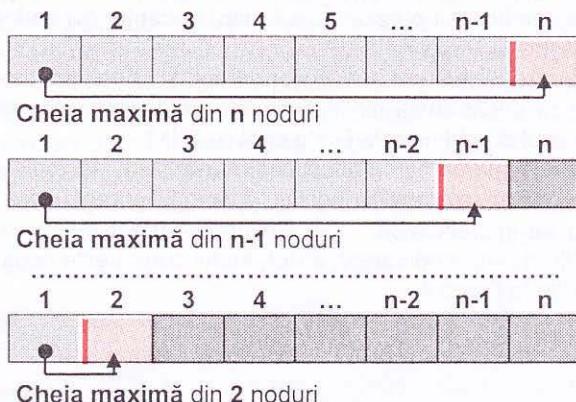
Complexitatea algoritmilor de prelucrare a ansamblurilor Heap

În **algoritmul de creare** a unui ansamblu Heap, se parcurge secvențial vectorul pentru adăugarea celor n elemente. Complexitatea parcurgerii secvențiale este $O(n)$. Pentru fiecare element adăugat la vector, cheia nodului trebuie adusă în poziția corespunzătoare din ansamblul Heap. Cazul cel mai defavorabil este atunci când cheia se propagă de pe nivelul pe care a fost adăugată, până la nivelul rădăcinii, numărul de comparații executate fiind egal cu înălțimea h a arborelui la acel moment (care ia valori de la 0 până la $\lg_2 n$), iar complexitatea algoritmului de propagare a unui nod de pe nivelul cu înălțimea h până în rădăcină va fi $O(\lg_2 n)$. Rezultă că, în algoritmul de creare, complexitatea este $O(n) \times O(\lg_2 n) = O(n \lg_2 n)$. **Algoritmul de inserare** a unui nod are complexitatea $O(\lg_2 n)$ – cazul cel mai defavorabil fiind atunci când cheia se propagă de pe ultimul nivel până în rădăcină. **Algoritmul de adăugare** a unui nod are complexitatea $O(\lg_2 n)$ – cazul cel mai defavorabil fiind atunci când cheia se propagă din rădăcină până la ultimul nivel.

2.8.6.3. Algoritmul HeapSort

Dacă un vector este organizat ca ansamblu Heap, nodurile vor fi extrase în ordinea precizată prin relația de ordine asociată arborelui. Această caracteristică a ansamblurilor Heap, stă la baza algoritmului de sortare HeapSort. Dacă fiecare element eliminat este adus după ultimul element din vectorul rămas după extragerea lui, în vectorul obținut, cheile vor fi ordonate invers decât prin relația de ordine prestabilită în ansamblul Heap. Astfel:

- Dacă vectorul era organizat ca un ansamblu **Heap Maxim**, va fi ordonat descrescător.
 → Dacă vectorul era organizat ca un ansamblu **Heap Minim**, va fi ordonat crescător.



Algoritmul este:

- PAS1.** Se creează vectorul care urmează să fie sortat ca un vector care implementează un ansamblu Heap.
- PAS2.** Se salvează lungimea logică a vectorului.
- PAS3.** Se initializează nodul curent cu ultimul nod (indicele i are valoarea n).
- PAS4.** Cât timp mai există noduri de extras din ansamblul Heap **execută**
- PAS5.** Se elimină nodul din arbore și se salvează în nodul curent.
 - PAS6.** Nodul curent devine ultimul nod din ansamblul rămas după eliminare (se decrementează cu 1 indicele nodului curent) și se revine la Pasul 4.

```
....//datele și structurile de date declarate global
void creare_heap() {...}
void afisare(){...}
int elibera() {...}
void HeapSort()
{int i,j=n;
 for (i=n;i>1;i--) v[i]=elibera();
 n=j;}
void main() {creare_heap(); HeapSort(); afisare(); cout<<endl;}
```

Complexitatea algoritmului de sortare

În **algoritmul de sortare** a unui ansamblu Heap se parcurge secvențial vectorul pentru a elimina cele n elemente. Complexitatea parcurgerii secvențiale este $O(n)$. Pentru fiecare nod se execută algoritmul de eliminare – care are complexitatea $O(\lg_2 n)$. Rezultă că, în algoritmul de sortare a unui vector organizat ca ansamblu Heap, complexitatea este $O(n) \times O(\lg_2 n) = O(n \cdot \lg_2 n)$. **Algoritmul de sortare** a unui vector folosind metoda HeapSort presupune organizarea vectorului ca un ansamblu Heap – complexitatea $O(n \cdot \lg_2 n)$, urmată de sortarea ansamblului Heap – complexitatea $O(n \cdot \lg_2 n)$. Rezultă că în algoritmul de sortare prin metoda HeapSort, complexitatea este $O(n \cdot \lg_2 n) + O(n \cdot \lg_2 n) = O(2 \cdot n \cdot \lg_2 n) = O(n \cdot \lg_2 n)$.

Tema



Se citesc din fișierul BAC.TXT mai multe siruri de numere naturale nenule, sfârșitul fiecărui sir fiind marcat de o valoare 0 (care nu se consideră că face parte din vreun sir). Știind că fiecare dintre siruri este un sir strict

crescător și că în total sunt cel mult 5000 de numere, scrieți un program care să realizeze afișarea tuturor numerelor nenule distincte din fișier, în ordinea crescătoare a valorilor lor.

- Descrieți două metode de rezolvare, una dintre metode fiind eficientă ca timp de executare.
- Scrieți un program corespunzător metodei eficiente descrise.

(Bacalaureat – Simulare 2004)

2.8.6.4. Aplicații practice

- Mai multe vehicule parcurg același traseu. Fiecare vehicul este caracterizat de un identificator, timpul de pornire, timpul de sosire și timpul de parcurgere a traseului (care se calculează din diferența celor doi timpuri). Scrieți un program care să memoreze aceste informații într-un ansamblu heap și care să asigure următoarele funcții: adăugarea unui nod (un vehicul care a sosit) și afișarea informațiilor despre vehicule, în ordinea crescătoare a timpului de parcurs.
- Produsul realizat de o secție a unei fabrici este cerut de mai mulți clienți. Fiecare client este caracterizat de nume, număr de telefon, data la care trebuie livrat produsul și cantitatea comandată. Cererile sunt onorate în ordinea calendaristică a datelor de livrare și în funcție de cantitatea disponibilă în stoc. Scrieți un program care să permită întărirea zilnică a acestei structuri de date, prin următoarele operații:
 - adăugarea de noi comenzi;
 - afișarea listei de comenzi care trebuie onorate în ziua respectivă și salvarea ei într-un fișier text (data calendaristică și stocul de produse la începutul zilei se citesc de la tastatură);
 - stocul la sfârșitul zilei, după ce au fost onorate comenziile.

Evaluare

Adevărat sau Fals:

- Arboarele este un graf fără cicluri, minimal cu această proprietate.
- Arboarele parțial de cost minim determină drumurile de lungime minimă dintre oricare două noduri dintr-un graf conex.
- În algoritmul lui Kruskal pentru construirea APM, se pornește de la un arbore parțial care conține nodul rădăcină.
- Într-un arbore cu rădăcină, implementarea cu referințe ascendente folosește doi vectori: unul pentru a memora nodurile terminale și altul pentru a memora părinții nodurilor terminale.
- Algoritmul de parcurgere în lățime a unui arbore cu rădăcină prelucrează informația în funcție de relațiile de subordonare dintre noduri.
- Arboarele binar este un arbore cu rădăcină, în care fiecare nod are cel mult doi succesiuni.
- Un arbore binar strict este un arbore echilibrat.
- Un arbore vid este un arbore în care nodul rădăcină nu are succesiuni.
- Într-un arbore binar de căutare, cheia oricărui nod este mai mare decât cheile succesiunilor lui.
- Pentru a sorta crescător un vector cu algoritmul HeapSort, vectorul trebuie să implementeze un arbore binar în care, pentru fiecare nod, cheia din succesorul stâng este

mai mică decât cheia din nod, iar cheia din succesorul drept este mai mare decât cheia din nod.

11. Într-un ansamblu Heap, cheile nodurilor trebuie să fie distincte.

Alegeți:

1. Orice graf neorientat cu n noduri și $n-1$ muchii:
 - a. este aciclic și neconex
 - b. este conex dacă și numai dacă este aciclic
 - c. este un arbore
 - d. este conex și conține un ciclu

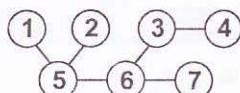
(Bacalaureat – Simulare 2003)
2. Numărul de muchii ale unui arbore cu 9 noduri este:
 - a. 9
 - b. 7
 - c. 8
 - d. 10
3. Numărul maxim de noduri terminale ale unui arbore liber cu 15 noduri este:
 - a. 15
 - b. 14
 - c. 12
 - d. 10
4. Numărul maxim de noduri terminale ale unui arbore cu rădăcină cu 15 noduri este:
 - a. 15
 - b. 14
 - c. 12
 - d. 10
5. Numărul maxim de noduri terminale ale unui arbore binar cu 15 noduri este:
 - a. 6
 - b. 7
 - c. 5
 - d. 4
6. Se consideră un arbore cu următoarele proprietăți: rădăcina este nodul 1 (considerat de nivel 1) și fiecare nod i (cu $1 \leq i \leq 3$) aflat pe nivelul j are ca descendenți nodurile $i+j$ și $i+2*j$. Nodurile i (cu $i > 3$) sunt noduri terminale (frunze). Stabilitățile care dintre vectorii următori este vectorul de tată (sau de predecesori) corespunzător arborelui:

a. 0 1 2 2 1 3 3	b. 0 -1 1 -1 -1 1 1
c. 0 1 1 2 3 2 3	d. 0 1 1 2 2 3 3

(Bacalaureat – Simulare 2003)

7. Se consideră arborele din figura alăturată. Care dintre noduri trebuie ales ca rădăcină astfel încât înălțimea arborelui să fie minimă.

- a. 7
- b. 3
- c. 5
- d. 6



(Bacalaureat – Sesiunea specială 2003)

8. Un arbore are 14 noduri. Atât rădăcina, cât și fiecare dintre nodurile neterminale au cel mult 3 descendenți direcți. Înălțimea maximă a arborelui (numărul maxim de muchii ce leagă rădăcina de o frunză) este:

- a. 4
- b. 13
- c. 5
- d. 3

(Bacalaureat – Sesiunea specială 2003)

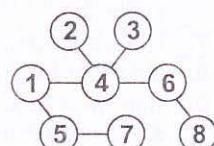
9. Memorarea unui arbore cu ajutorul matricei de adiacență a grafului este o metodă:

- a. eficientă
- b. neeficientă
- c. incorrectă
- d. recomandabilă

(Bacalaureat – Sesiunea specială 2003)

10. Se consideră arborele din figura A28. Definim înălțimea arborelui ca fiind numărul maxim de muchii care formează un lanț care unește rădăcina de un nod terminal. Care este înălțimea maximă a arborelui cu rădăcină ce se poate obține alegând ca rădăcină unul dintre noduri?

- a. 5
- b. 3
- c. 6
- d. 7



(Bacalaureat – Sesiunea iunie-iulie 2003)

11. Se consideră un arbore cu rădăcină în care orice nod care nu este terminal are exact 3 descendenți direcți. Atunci, numărul de noduri terminale (frunze) poate fi:

a. 15

b. 24

c. 2

d. 14

(Bacalaureat – Sesiunea iunie-iulie 2003)

12. Se consideră un arbore în care rădăcina este considerată de nivelul 1 și orice nod de pe nivelul i are exact $i+1$ descendenți direcți, cu excepția nodurilor de pe nivelul 4 care sunt noduri terminale. Numărul de noduri terminale (frunze) sunt:

a. 24

b. 120

c. 6

d. 30

(Bacalaureat – Sesiunea august 2003)

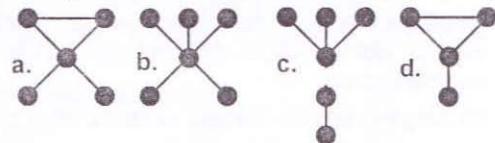
13. Se consideră un arbore. Despre un lanț care unește două noduri distințe n_1 și n_2 se poate afirma:

a. este unic oricare ar fi n_1 și n_2 b. este unic dacă și numai dacă n_1 sau n_2 este terminalc. nu este unic oricare ar fi n_1 și n_2 d. este unic dacă și numai dacă n_1 sau n_2 este rădăcină

(Bacalaureat – Sesiunea august 2003)

14. Stabilități care dintre grafurile alăturate este arbore:

(Bacalaureat – Simulare 2004)



Următorii 4 itemi se referă la arborele cu rădăcină cu 8 noduri, cu muchiile [1,7], [1,8], [3,1], [3,4], [3,5], [4,2], [4,6] și cu rădăcina în nodul cu eticheta 3.

15. Vectorul tată al arborelui este:

a. 3 4 0 3 3 4 1 2

b. 3 3 0 4 3 4 1 1

c. 3 4 0 3 3 4 1 1

d. 3 4 3 3 4 1 1 0

16. Numărul de noduri terminale ale arborelui este:

a. 6

b. 4

c. 5

d. 3

17. Înălțimea maximă pe care o poate avea, schimbându-se nodul, rădăcină este:

a. 6

b. 5

c. 4

d. 3

18. Dacă se ia ca rădăcină nodul cu eticheta 2, înălțimea arborelui este:

a. 5

b. 2

c. 4

d. 3

19. Numărul de componente conexe care se obțin dacă se elimină muchiile care au o extremitate în nodul cu eticheta 3 este:

a. 3

b. 4

c. 2

d. 1

Următorii 5 itemi se referă la arborele cu rădăcină cu 8 noduri care are vectorul tată: 2 0 2 1 1 1 3 3.

20. Numărul de noduri terminale ale arborelui este:

a. 3

b. 4

c. 5

d. 2

21. Numărul de lanțuri de lungime 3 este:

a. 3

b. 4

c. 5

d. 6

22. Numărul de lanțuri de lungime 2 care pornesc din rădăcină este:

a. 2

b. 4

c. 3

d. 5

23. Numărul de nivele ale arborelui este:

a. 2

b. 3

c. 4

d. 5

24. Fiile nodului 3 sunt:

a. 4, 5, 6

b. 1, 2

c. 7, 8

d. 2, 7, 8

25. Un arbore binar strict cu 4 noduri terminale are în total

a. 5 noduri

b. 7 noduri

c. 9 noduri

d. 10 noduri

26. Un arbore binar strict are:

- a. un număr par de noduri b. un număr impar de noduri
 c. un număr par de muchii d. un număr impar de muchii

27. Se consideră arborele din figura alăturată. Stabilită ce fel de arbore nu este:

- a. arbore binar b. arbore de căutare
 c. ansamblu Heap d. arbore strict

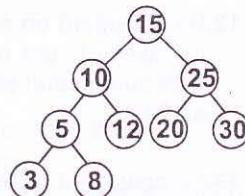
28. Se consideră arborele din figura de la problema anterioară. Stabilită ce fel de arbore nu este:

- a. arbore echilibrat b. arbore complet
 c. arbore aproape complet d. arbore strict

29. Un arbore binar strict cu 4 noduri terminale are în total:

- a. 4 muchii b. muchii c. 8 muchii d. 9 muchii

Următorii 5 itemi se referă la arborele binar din figura alăturată; r este un pointer către rădăcină, iar p și q, doi pointeri către un nod oarecare:



30. Alegeti varianta corectă pentru a afișa cheia 15:

- a. `cout<<r->d->d>->info;` b. `cout<<r->s->s>->info;`
 c. `p=r->s->d; cout<<p->d->info;` d. `p=r->s->d; cout<<p->info;`

31. Dacă se execută instrucțiunea: `p=r->s->d->d;` pointerul p va indica cheia:

- a. 21 b. 20 c. 18 d. 25

32. Următoarea secvență de instrucțiuni:

`p=r->d->d; p=p->s; q=new nod; q->info=5; q->s=NULL; q->s=NULL; p->s=q;`
 realizează:

- a. adaugă nodul cu cheia 5 ca succesor stâng al nodului cu cheia 30;
 b. adaugă nodul cu cheia 5 ca succesor drept al nodului cu cheia 30;
 c. adaugă nodul cu cheia 5 ca succesor stâng al nodului cu cheia 19;
 d. adaugă nodul cu cheia 5 ca succesor drept al nodului cu cheia 19;

33. Următoarea secvență de instrucțiuni:

`for (p=r; p->s!=NULL; p=p->s); q=new nod;
 q->info=20; q->s=NULL; q->s=NULL; p->s=q;`
 realizează:

- a. adaugă nodul cu cheia 20 ca succesor stâng al nodului cu cheia 25;
 b. adaugă nodul cu cheia 20 ca succesor drept al nodului cu cheia 25;
 c. adaugă nodul cu cheia 20 ca succesor stâng al nodului cu cheia 21;
 d. adaugă nodul cu cheia 20 ca succesor drept al nodului cu cheia 21;

34. Următoarea secvență de instrucțiuni:

`for (p=r; p->d->d!=NULL; p=p->d); q=p->d; p->d=NULL; dispose q;`
 realizează:

- a. elimină nodul cu cheia 21; b. elimină nodul cu cheia 18;
 c. elimină nodul cu cheia 25; d. elimină nodul cu cheia 30;

Miniproiecte:

- Pentru evidența elevilor din clasă, se păstrează următoarele informații: numărul de ordine din catalog, numele și prenumele, mediile semestriale și media anuală la disciplina informatică. Realizați o aplicație care să asigure următoarele funcții:
 a. adăugarea unui nou elev;

- b. eliminarea unui elev;
 - c. modificarea mediei unui elev;
 - d. afişarea elevilor cu mediile (semestriale, respectiv anuale), în ordinea descrescătoare a mediilor;
 - e. afişarea elevilor cu mediile (semestriale, respectiv anuale) cuprinse între două valori care se citesc de la tastatură.
2. Într-o bibliotecă, pentru fiecare carte se păstrează următoarele informații: codul cărții, titlul, autorul, numărul total de exemplare și numărul de exemplare disponibile la un moment dat. Realizați o aplicație care să asigure următoarele funcții:
- a. regăsirea unei cărți, după autor, pentru a afișa numărul de exemplare disponibile;
 - b. adăugarea unui nou titlu;
 - c. adăugarea de noi exemplare, pentru titlurile existente;
 - d. modificarea numărului total de exemplare (în cazul distrugerii unora dintre ele);
 - e. modificarea numărului de exemplare disponibile, în cazul în care o carte este împrumutată sau înapoiată;
 - f. afişarea cărților pentru care nu mai există exemplare disponibile;
 - g. afişarea în ordine alfabetică a autorilor.
3. Pentru fiecare articol dintr-o magazie, se păstrează următoarele informații: codul articolului (este unic pentru fiecare articol), denumirea articolului, unitatea de măsură și cantitatea din stoc. Realizați o aplicație pentru gestionarea magaziei – care să asigure următoarele funcții:
- a. adăugarea unui nou articol;
 - b. ştergerea unui articol;
 - c. introducerea de la tastatură a cantității intrate, dintr-un articol precizat prin cod, și actualizarea stocului;
 - d. introducerea de la tastură a cantității ieșite, dintr-un articol precizat prin cod, și cu actualizarea stocului;
 - e. afişarea articolelor care au stocul pe 0;
 - f. afişarea stocului unui articol al cărui cod se introduce de la tastatură;
 - g. afişarea, în ordinea codurilor, a stocurilor din fiecare articol.

Cuprins

1. Tehnici de programare	3
1.1. Analiza algoritmilor	3
1.2. Metode de construire a algoritmilor	5
1.3. Metoda backtracking	6
1.3.1. Descrierea metodei backtracking	6
1.3.2. Implementarea metodei backtracking	10
1.3.3. Probleme rezolvabile prin metoda backtracking	14
1.3.3.1. Generarea permutărilor	15
1.3.3.2. Generarea produsului cartezian	17
1.3.3.3. Generarea aranjamentelor	20
1.3.3.4. Generarea combinațiilor	22
1.3.3.5. Generarea tuturor partițiilor unui număr natural	24
1.3.3.6. Generarea tuturor partițiilor unei multimi	27
1.3.3.7. Generarea tuturor funcțiilor surjective	28
1.3.3.8. Problema celor n dame	30
1.3.3.9. Parcursarea tablei de săh cu un cal	31
1.3.3.10. Generarea tuturor posibilităților de ieșire din labirint	34
1.4. Metoda „Divide et Impera“	40
1.4.1. Descrierea metodei „Divide et Impera“	40
1.4.2. Implementarea metodei „Divide et Impera“	41
1.4.3. Căutarea binară	48
1.4.4. Sortarea rapidă (QuickSort)	50
1.4.5. Sortarea prin interclasare (MergeSort)	53
1.4.6. Problema turnurilor din Hanoi	54
1.4.7. Generarea modelelor fractale	56
1.5. Metoda greedy	59
1.5.1. Descrierea metodei greedy	59
1.5.2. Implementarea metodei greedy	61
1.6. Metoda programării dinamice	70
1.6.1. Descrierea metodei programării dinamice	70
1.6.2. Implementarea metodei programării dinamice	73
1.7. Compararea metodelor de construire a algoritmilor	83
Evaluare	85
2. Implementarea structurilor de date	90
2.1. Tipuri de date specifice pentru adresarea memoriei	90
2.2. Tipul de dată pointer	91
2.2.1. Declararea variabilei de tip pointer	92
2.2.2. Constante de tip adresă	92
2.2.3. Operatori pentru variabile de tip pointer	94
2.2.3.1. Operatorii specifici	94
2.2.3.2. Operatorul de atribuire	97
2.2.3.3. Operatorii aritmetici	100
2.2.3.4. Operatorii relaționali	101
2.3. Tipul de dată referință	102
2.4. Alocarea dinamică a memoriei	106
2.5. Clasificarea structurilor de date	109
2.6. Lista liniară	113
2.6.1. Implementarea dinamică a listelor în limbajul C++	115
2.6.2. Clasificarea listelor	117
2.6.3. Algoritmi pentru prelucrarea listelor simplu înlăntuite	118
2.6.3.1. Inițializarea listei	118
2.6.3.2. Crearea listei	118

2.6.3.3. Adăugarea primului nod la listă	118
2.6.3.4. Adăugarea unui nod la listă.....	119
2.6.3.5. Parcurgerea listei	121
2.6.3.6. Căutarea unui nod în listă	121
2.6.3.7. Eliminarea unui nod din listă	122
2.6.3.8. Eliberarea spațiului de memorie ocupat de listă	123
2.6.3.9. Liste ordonate	124
2.6.3.10. Prelucrarea listelor simplu înlățuite	127
2.6.4. Algoritmi pentru prelucrarea listelor circulare simplu înlățuite.....	137
2.6.4.1. Crearea listei	137
2.6.4.2. Parcurgerea listei	138
2.6.4.3. Eliminarea unui nod din listă	138
2.6.5. Algoritmi pentru prelucrarea listelor dublu înlățuite.....	140
2.6.5.1. Adăugarea primului nod la listă	140
2.6.5.2. Adăugarea unui nod la listă.....	140
2.6.5.3. Parcurgerea listei	141
2.6.5.4. Eliminarea unui nod din listă	141
2.6.6. Algoritmi pentru prelucrarea stivelor	145
2.6.6.1. Inițializarea stivei.....	145
2.6.6.2. Adăugarea unui nod la stivă.....	146
2.6.6.3. Extragerea unui nod din stivă.....	146
2.6.6.4. Prelucrarea stivei.....	146
2.6.7. Algoritmi pentru prelucrarea cozilor	148
2.6.7.1. Inițializarea cozii	149
2.6.7.2. Adăugarea unui nod la coadă	149
2.6.7.3. Extragerea unui nod din coadă	149
2.6.7.4. Prelucrarea cozii.....	149
2.6.8. Aplicații practice	151
Evaluare	152
2.7. Graful	159
2.7.1. Definiția matematică a grafului.....	159
2.7.2. Graful neorientat	160
2.7.2.1. Terminologie	160
2.7.2.2. Gradul unui nod al grafului neorientat	162
2.7.2.3. Sirul grafic	164
2.7.3. Graful orientat	165
2.7.3.1. Terminologie	165
2.7.3.2. Gradele unui nod al grafului orientat	167
2.7.4. Reprezentarea și implementarea grafului	169
2.7.4.1. Reprezentarea prin matricea de adiacență	169
2.7.4.2. Reprezentarea prin matricea de incidentă	175
2.7.4.3. Reprezentarea prin lista muchiilor (arcelor)	182
2.7.4.4. Reprezentarea prin lista de adiacență (listele vecinilor).....	187
2.7.4.5. Aplicații practice	197
2.7.5. Grafuri speciale	201
2.7.5.1. Graful nul.....	201
2.7.5.2. Graful complet	201
2.7.6. Grafuri deriveate dintr-un graf	203
2.7.6.1. Graful parțial	203
2.7.6.2. Subgraful	207
2.7.6.3. Graful complementar	211

2.7.7. Conexitatea grafurilor.....	213
2.7.7.1. Lantul	213
2.7.7.2. Ciclul	218
2.7.7.3. Drumul.....	220
2.7.7.4. Circuitul	223
2.7.7.5. Graful conex.....	225
2.7.7.6. Graful tare conex.....	228
2.7.4.7. Aplicații practice	235
2.7.8. Parcurgerea grafului	235
2.7.8.1. Parcurgerea în lățime – Breadth First	236
2.7.8.2. Parcurgerea în adâncime – Depth First	240
2.7.8.3. Aplicații practice	246
2.7.9. Graful ponderat.....	246
2.7.9.1. Definiția grafului ponderat	246
2.7.9.2. Matricea costurilor	247
2.7.9.3. Algoritmi pentru determinarea costului minim (maxim)	249
2.7.9.4. Aplicații practice	255
2.7.10. Grafuri speciale.....	256
2.7.10.1. Graful bipartit.....	256
2.7.10.2. Graful hamiltonian	260
2.7.10.3. Graful eulerian.....	263
2.7.10.4. Graful turneu	268
2.7.10.5. Aplicații practice	271
Evaluare	272
2.8. Arboarele.....	281
2.8.1. Arborele liber	281
2.8.1.1. Definiția arborelui liber.....	281
2.8.1.2. Proprietățile arborilor liberi	281
2.8.2. Arborele parțial	283
2.8.2.1. Definiția arborelui parțial.....	283
2.8.2.2. Definiția arborelui parțial de cost minim	283
2.8.2.3. Algoritmi de determinare a arborelui parțial de cost minim	285
2.8.2.4. Aplicații practice	292
2.8.3. Arborele cu rădăcină	292
2.8.3.1. Definiția arborelui cu rădăcină	292
2.8.3.2. Implementarea arborelui cu rădăcină	297
2.8.3.3. Algoritmi pentru parcugerea unui arbore cu rădăcină	299
2.8.3.4. Aplicații practice	304
2.8.4. Arborele binar	305
2.8.4.1. Definiția arborelui binar	305
2.8.4.2. Implementarea arborelui binar	305
2.8.4.3. Algoritmi pentru parcugerea unui arbore binar.....	312
2.8.4.4. Aplicații practice	315
2.8.5. Arborele binar de căutare	315
2.8.5.1. Definiția arborelui binar de căutare	315
2.8.5.2. Algoritmi pentru prelucrarea unui arbore binar de căutare	315
2.8.5.3. Aplicații practice	322
2.8.6. Ansamblul Heap	323
2.8.6.1. Definiția ansamblului Heap	323
2.8.6.2. Algoritmi pentru prelucrarea unui ansamblu Heap	324
2.8.6.3. Algoritmul HeapSort	327
2.8.5.4. Aplicații practice	329
Evaluare	329

ISBN 973 - 30 - 1567 - 9
ISBN 978 - 973 - 30 - 1567 - 3

ISBN 973 - 30 - 1567 - 9

